2001-05-02

Garbage Collection for Java Distributed Objects

Andrei Arthur Dancus
Worcester Polytechnic Institute

GARBAGE COLLECTION FOR JAVA DISTRIBUTED OBJECTS

by

Andrei A. Dãncus

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements of the

Degree of Master of Science

in

Computer Science

by

_____

Andrei A. Dãncus

Date: May 2nd, 2001

Approved:

_____

Dr. David Finkel, Advisor

_____

Dr. Mark L. Claypool, Reader

_____

Dr. Micha Hofri, Head of Department

Abstract

We present a distributed garbage collection algorithm for Java distributed objects using the object model provided by the Java Support for Distributed Objects (JSDA) object model and using weak references in Java. The algorithm can also be used for any other Java based distributed object models that use the stub-skeleton paradigm. Furthermore, the solution could also be applied to any language that supports weak references as a mean of interaction with the local garbage collector. We also give a formal definition and a proof of correctness for the proposed algorithm.

Acknowledgements

I would like to express my gratitude to my advisor Dr. David Finkel, for his encouragement and guidance over the last two years. I also want to thank Dr. Mark Claypool for being the reader of this thesis. Thanks to Radu Teodorescu, co-author of the initial JSDA project, for reviewing portions of the JSDA Parser.

Table of Contents

1. Introduction

Garbage collection – the automatic reclamation of heap-allocated data after its last use in a program – is no longer considered a luxury feature of programming languages, and Java is the best example. There are, of course, exceptions – some real-time applications – but the myth of unacceptable overhead introduced by garbage collection is not that strong nowadays.

Like single-machine applications, distributed applications also need to be supported by garbage collection. The Java language is well suited to the creation of various distributed object models because of its orientation towards simplicity in network access. JSDA (Java Support for Distributed Application – [DTH99]) is such a model.

Our initial goal was to determine whether the Java language offers the necessary mechanisms for building a distributed garbage collector. This was not obvious, since the language provides a very limited degree of interaction with the local collector.

The Reference Objects (java.lang.ref) API seemed to provide us with enough of the functionality that we needed, so we started designing an algorithm using this new – introduced by version 1.2 – Java features. We proved the correctness of this algorithm and implemented it in JSDA.

This document contains two main parts. Chapter 2 provides some background and related work in the fields of distributed programming and garbage collection. Chapter 3 starts by putting the two concepts together and describing what the issues are; then we present the Reference Objects API and describe how it can be used as a tool for building our algorithm, for which a formal description and proof are given; sample scenarios are

also presented in order to show how our approach avoids race conditions and to explain the rationale behind the rules that define the algorithm. We also describe why and how particular design decisions were implemented.

Chapter 4 summarizes presents the conclusions of our work – it summarizes our achievements and it presents some possible directions for future work.

2. Background and Related Work

2.1 Distributed Programming. JSDA

Distributed Programming is a relatively new field of Computer Science. The main reason for that is that until about two decades ago, computer networks were very rare even in the academic environments. Once building a network became cheaper, people (both the research community and the industry) started to look for solutions for making programming in a distributed environment easier.

One of the first few steps towards providing some abstraction at the programming level in a distributed system was made by Andrew Birell and Bruce Nelson about 20 years ago. They proposed and implemented the RPC (Remote Procedure Calls) model, which allows programmers to invoke procedures on remote machines as if they were local [BN91]. The transparency of remote calls was made possible by the use of *client stubs*. A stub is piece of code (automatically generated by a software tool) that is responsible with *marshalling* the request (along with the parameters) and sending it over the network. On the server side, a *server stub* will do the reverse process, it will invoke the local procedure, and then it will send the results back to the caller.

This model proved to be successful and other companies created their own RPC modules. Sun Microsystems implemented their own RPC library (SunRPC, in 1985) [C91]; this is still in use today as an underlying mechanism for the NFS file system.

However, it became clear that – although useful – RPC was still a low-level tool for distributed programming and that a more abstract approach was needed. At about the same, research on distributed operating systems was trying to attack a similar problem – how to distribute tasks in a network of computers running a single operating system hiding the distribution aspects.

Emerald [BHJL86], *Linda* [Lin], *Orca* [Orc] are some of the languages that were designed to meet the requirements of a distributed (and parallel – the last two mentioned) programming languages. However, they illustrate very different approaches. Emerald was developed for object-based systems, and allowed migration for some of its objects. Linda is based on the very original (and elegant) idea of *tuple-space*. The tuple-space is essentially an abstraction for shared memory; in Linda, processes interact by using the tuple-space as a "bag" in/from which they can put/pick up *tuples*. Passive tuples only contain data, while active data also contain operations that can generate new tuples. Orca was developed for the Amoeba distributed operating system.

CORBA

None of these languages became popular but a new approach was proposed. Instead of using a single language to write distributed applications, CORBA (Common

Object Request Broker Architecture) [Cor] was proposed as a standard for connecting existing objects, written in any language. An IDL (Interface Definition Language) is used in order to define wrappers around objects, so that they can interact in a standard way. A client application can access a CORBA object via a local IDL stub that talks to an IDL skeleton (on the server-side) via an ORB (Object Request Broker). Dynamic invocation of objects is also possible – in this case stubs and skeletons are not needed; clients can invoke requests on any object without having compile-time knowledge of the object's interface. Figure 2.1.2 shows how the components of CORBA work together. An important component (on the server side) is the Object Adapter, whose responsibilities include: object registration, reference generation, server process / object activation, request demultiplexing, object upcalls.

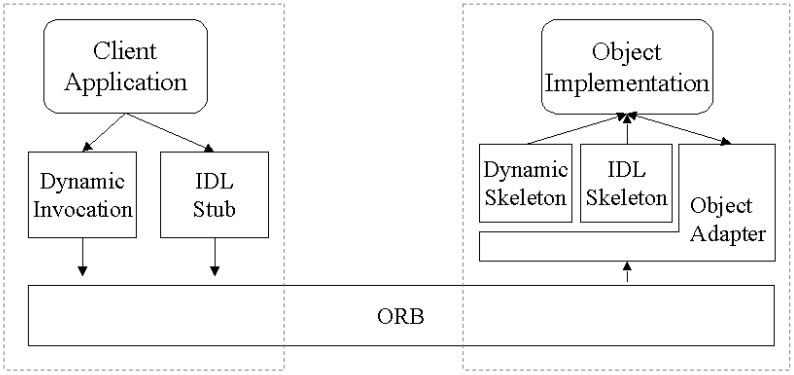


Figure 2.1.2: The Object Request Broker Architecture

DCOM (Distributed Component Object Model Architecture) is Microsoft's technology [DCOM] for access to remote objects. Although it can be – in theory – implemented on any platform, DCOM is almost exclusively used on Windows systems, which is a serious drawback if cross-platform interoperability is required.

Java, Java RMI, and Java-based approaches

When Java was released in 1995, a revolution started. Although Java is not a true distributed programming language, it makes distributed programming easy by incorporating a lot of useful network-related libraries (packages) in its core API. It also provides mechanisms for dynamic class loading and for dynamic discovery of objects' capabilities (the "reflection" package). These features along with its platform-independence and simplicity made it *the* language in the heavily networked, heterogeneous world of the 1990s.

In addition to its low level packages (for socket programming), Java provides – since version 1.2 (1997) – a mechanism for remote method invocation (RMI). The RMI model [RMI98] provides a higher level of abstraction and gives Java the flavor of a distributed language. It is suited for client-server architectures and uses the same idea (as RPC, CORBA) of stubs and skeletons. Figure 2.1.2 illustrates the RMI mechanism:

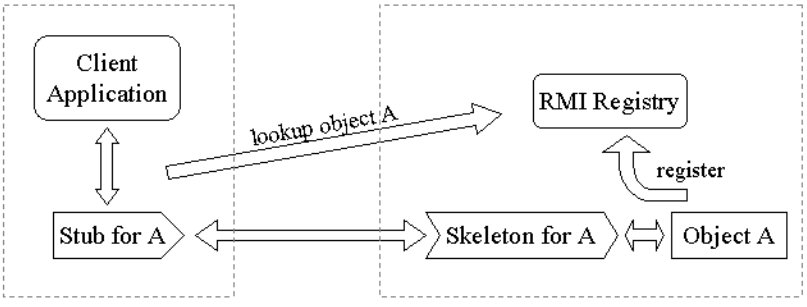


Figure 2.1.2: Java Remote Method Invocation Architecture

There are a lot of approaches that use Java for distributing computing, and – naturally – many of them use RMI. Some are tools for implementing agent technologies (Voyager [Vo], Odyssey), others propose a modified RMI mechanism (JavaParty, FarGo [Far]) and there are also a couple approaches that use the idea of tuple spaces that was introduced by Linda (JavaSpaces [JSp] , Tspaces).

Although some of these have been used as tools for developing distributed applications for specific fields (e.g. Voyager), none of them is widely used. We believe that one of the main reasons for that is that they require the Java programmer to understand a new programming model and to become familiar with (usually a lot of) new classes; Voyager for example contains 19 packages. Moreover, they cannot be regarded as an attempt to make Java a distributed language, since they do not propose extensions at the language level.

JSDA

JSDA (Java Support for Distributed Application) [DTH99] is an approach for distributed programming that tries to focus on simplicity and ease of use, without affecting flexibility. Our primary goal when designing JSDA was to provide distribution capabilities at the language level. This could be done either by rewriting the JVM (Java Virtual Machine) – one such approach is used by [AFT99] – or by adding a pre-processing step for parsing the applications written in a Java "distributed style". JSDA falls in the second category, which has the advantage of portability on any system that

has a standard JVM and the JSDA platform (which is entirely written in Java). The remaining of this section provides a brief description of JSDA architecture and concepts.

JSDA consists of 2 main components: the Parser and the Kernel (the runtime):

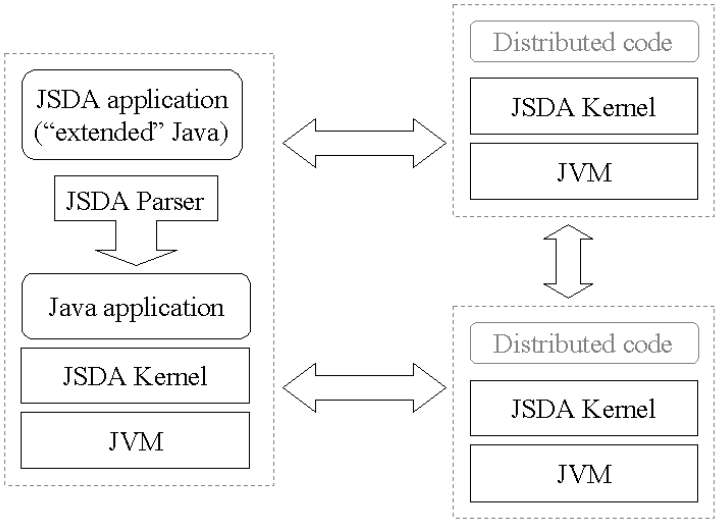


Figure 2.1.3: The JSDA approach

The JSDA distributed object model

The distributed object model that JSDA relies on has similarities with RMI and CORBA. The idea is that a distributed object consists of a *master object* (that resides on a machine that we call the *owner* of that object) plus *ambassador objects (stubs)* that provide access to the master object from any host in the distributed system.

One of the key differences between JSDA and RMI/CORBA is that master objects are associated to physical machines *at runtime* in JSDA. This is a natural consequence of the fact that JSDA tries to encourage and simplify distribution at the

language level. On the other hand, RMI/CORBA are better suited to the idea of server objects that offer specific services on designated machines.

The association master object – physical machine is done in the initialization step of the JSDA Kernel. Each machine/user that is eager to participate in a particular distributed application "logs on" to the server host – the host which contains the JSDA configuration file and whose JSDA Kernel is responsible with the launch of the distributed application. The login simply consists of a symbolic name that will be associated by the JSDA Kernel with the client's physical address. An example of how this works is presented later in this section.

An ambassador object (stub) has several key-roles:

1. It provides read/write access to the fields of the master object
2. It can execute methods of the master object on the local machine
3. It can initiate the execution for methods of the master object on any other machine
4. It provides synchronization features using the master object as a synchronization object

A distributed object consists of the sum of all its stubs plus one master, all of them having in common a globally unique ID.

Master/ambassador objects for a Java class are instances of the same (distributed version of the) class. It depends on the way one instance of the class in constructed whether that object becomes master or ambassador. A special field – named *owner* – in each distributed class specifies the name of the machine where the master object was built, making the difference between master and ambassador. This approach gave us the

possibility to deal with distributed objects in a homogenous way. This means that the code generated by JSDA Parser does not distinguish between ambassadors and masters; its kernel's job to decide whether a remote access is necessary or not and to take the necessary actions.

The JSDA directives

To keep the programming model as simple as possible, only two directives were added to the standard Java; their meaning is to force the execution of the following instruction (or block of instructions) on the specified host(s).

The two types of directives accepted are:

- **on <machine_name>** - this directive will force the execution of the following instruction/block on machine <*machine_name*>
- **foreach <machine_class_name> -** this directive will force the execution of the following instruction/block on all machines belonging to class <*machine_class_name*>

For example, the following sequence will display a message on all the hosts that belong to the class `audience`:

```
/* #[ foreach audience ]# */
   System.out.println("hello");
```

The following code will allocate a `ClassX` object on host `HostA`

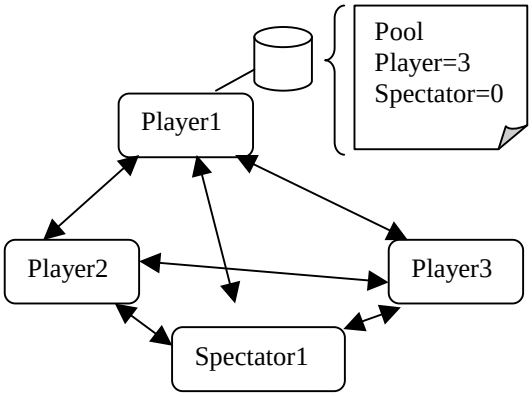
```
/* #[ on HostA ]# */
{ System.out.println(“Hello, I’ll eat some of your memory”);
  ClassX obj = new ClassX(params);
  System.out.println(“I’m done. I’ll go home now”;
}
```

Note that the JSDA directives are embedded in Java comments. The reason for choosing this syntax was to give the programmer the means to run the code, without modification, on a single machine for debugging purposes.

A sample scenario for a distributed pool game (with players and spectators using different machines) will require the following steps:

A. Create a configuration file containing the classes of machines and the name of the class that contains the code for the game.
B. Run the JSDA Parser in order to generate the distributed application
C. Launch the JSDA Kernel
D. Every machine that wants to join the game logs onto the server (using JSDA Kernel) submitting the machine class name
E. After the number of logged hosts in each class becomes equal with a minimum value (specified in the configuration file for each class), the distributed application starts.

Before the application starts, the JSDA Kernel creates TCP connections between each machine and all the other machines. A decentralized system is created, allowing code propagation between any two machines in the system. Figure 2.1.4 presents a scenario (for the distributed pool game). Player 1 is the host that contains the configuration file; it first runs the parser and then starts the distributed application.



	Configuration file
	TCP connection
Player1-3	Hosts belonging to machine class *Player*
Spectator1	Host belonging to machine class *Spectator*

Figure 2.1.4: JSDA sample scenario

The JSDA Kernel implements mechanisms for multithreading in the distributed system (we defined the term *distributed thread*), caching, synchronization. However, there are aspects like garbage collection and load balancing that were not addressed in [DTH99].

2.2 Garbage Collection

Garbage collection (GC) is the automatic reclamation of heap-allocated storage after its last usage by a program.

In the languages that support dynamic data allocation, deallocation has been traditionally done explicitly, at the language level (for instance, the *freemem* instruction is used in the C language). Besides simplifying the programmer's job, garbage collection has several important advantages. Jones ([JL96]) gives 3 primary reasons in favor of garbage collection:

- **Language requirements**. Functional languages (e.g. Lisp) have unpredictable execution orders and explicit deallocation is often impossible; garbage collection is mandatory.
- **Problem requirements.** A good example [BC92] is the following: suppose a general stack data type is to be implemented in C as a linked list; if the data may be pushed on two stacks, how should a *Pop* command behave in terms of deallocation? Since it is possible to push pointers to the *same data* on both stacks, we cannot simply free the memory every time a *Pop* is executed. Some convention is required for deallocation even for such a simple abstraction. This will either complicate the interface of the stack, reduce its applicability or force unnecessary copying.
- **Software engineering issues.** One common requirement for "good" software is encapsulation. Complex object-oriented applications consist of components which communicate through clearly defined interfaces. Programmer-controlled storage management inhibits this modularity, and most modern OO languages (Smalltalk,

Eiffel, Java, Dylan) are supported by garbage collection. Garbage collectors have also been written for uncooperative languages like C and C++.

One of the myths about garbage collection is that incurs unacceptable overhead. There is obviously a certain amount of extra work required, but it has been proven that in most of the cases this is acceptable. (There are, of course, some real-time applications for which this is not acceptable.) Java is probably the most popular example, but is noticeable that even languages used for systems programming such as Modula-2+ and Modula-3 are supported by garbage collection. In general, the typical execution overhead introduced by garbage collection ranges between a few percent to 20 percent according to [JL96], with 10 percent being considered unreasonable for a well-implemented system [W94].

2.2.1 Terminology

The values that a program can manipulate directly are those held in processor registers, those on the program stack and held in global variables. Such locations containing references to heap data form the **roots** of the computation.

An individually allocated piece of data in the heap is called **object**, **cell** or **node**. An object in the heap is called **live** (or **reachable**) if its address is held in a root or there is a pointer to it held in another live heap node.

Objects that become unreachable during execution are called **garbage**.

There are two main properties that ensure the correctness of a GC algorithm:

1. **Liveness –** All objects that are garbage will be collected eventually.
2. **Safety** – No live objects will be collected.

Usually there is a need to make a distinction between the garbage collector and the part of the program that does 'useful' work. We will use the term **mutator** for the user program (following Dijkstra's terminology [DL+78]), since – as far as the collector is concerned – its job is to change (or mutate) references among objects).

For distributed environments, we use the terms **space**, **node** or **host** to refer to each individual application that runs in a separate memory space and is usually capable of running its own local garbage collector.

An object (created on a machine called its **owner**) is accessible in a distributed system via a **stub**, **exit item**, or **surrogate** (on a client node), and via an **entry item** or **skeleton** (on the owner). When we talk about stubs/skeletons in RPC, CORBA, RMI etc. we are primarily interested in marshalling/unmarshalling remote requests. In the case of distributed garbage collection, the stub/skeletons are interesting from a different point of view – they can be used for storing and/or propagating additional information about inter-object references.

A **cycle** consists of a group of objects that refer each other, but none of them is accessible from the application (they are all garbage). If the group spans over multiple spaces, then we are dealing with a **distributed cycle**.

A **conservative** garbage collector is one that does not enforce immediate reclamation of unreachable data. This is usually done for performance reasons, but it results in temporary **floating garbage**.

2.2.2 Issues in Garbage Collection

Some desirable properties of an ideal garbage collector are:

- Completeness – all objects (ideally including components of **cycles)**, that are garbage at the start of a collection cycle should be reclaimed by its end
- Concurrency – neither mutator nor collector should be suspended; distinct distributed collection processes should run concurrently. Concurrency is further discussed in the next paragraph.
- Efficiency – time and space costs should be minimal

Depending on the technique used, some of these features will be hard (or even impossible) to accomplish.

Depending on the degree of interaction between the mutator and the collector, we can divide the garbage collection techniques in three categories:

- Sequential – stop-and-collect algorithms. The mutator has to stop whenever the collector executes.
- Incremental – incremental collectors do not suspend the mutator while garbage collector completes. However, an incremental collector will still pause the mutator

(but just for a small period of time) at each step of the collection algorithm. An incremental collector can be called **real-time** collector if the worst-case pause-times are bounded by problem-specific constants.

- Concurrent – these collectors have been developed in order to run on multiprocessor architectures (but they can be easily adapted to serial machines). The mutator and the collector run separate processes (and they still need to synchronize their actions).

2.2.3 Approaches for uniprocessor garbage collection

This section presents the existing non-distributed garbage-collection techniques ([JL96], [W94]). Distributed collectors extend these mechanisms and augment them with new features in order to make them suited to distributed environments.

2.2.3.1 Reference counting

Reference counting algorithms are based on the following idea: each cell maintains a counter, which represents the number of cells that reference this one. Each time a reference to a cell is created its counter is incremented, and each time a reference to it is deleted the counter is decremented. When the counter becomes zero, the object may be safely reclaimed as garbage.

Consider the example in Figure 2.2.3.1 which shows the counter for each object. Object D has its counter equal to 2 because there are 2 other objects holding references to it, while the counter for object E is 0, therefore this object can be garbage collected.

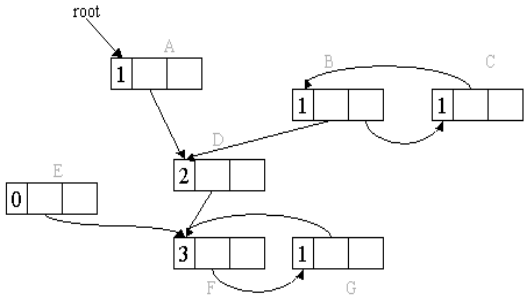


Figure 2.2.3.1 Reference Counting

This approach is naturally incremental for all operations except for the deletion of the last pointer to an object.

A variant of reference counting (sometimes called **reference listing**) uses a simple data structure (instead of a counter) to keep track of the references held to an object and also to provide information about *who* keeps references to it. This approach is more robust (especially in a distributed environment; for example duplicate messages cannot result in premature collection of (live) objects).

If reference listing will be used for the example above, then the node F will look like in Figure 2.2.3.1b:

Figure 2.2.3.1b Reference listing - the counter is replaced by a list

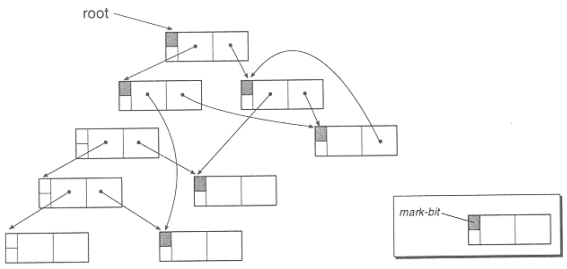
The main drawback of reference counting-based approaches is that they cannot reclaim cycles – note that in the example presented before (Figure 2.2.3.1) B and C have non-zero counters, therefore they will not be garbage collected although they are unreachable! There has been some work in this direction ([Ch84]], [L92]), but the only solution found so far is to periodically run a mark-sweep collector, whose only purpose is to detect cycles of garbage. Mark-sweep collectors are described in the next subsection.

2.2.3.2 Mark and sweep

Mark and sweep collectors run in two phases:

1. **Mark Phase** – distinguish the live objects from the garbage.

   This is done by tracing – starting at the root and actually traversing the graph of pointer relationships – usually by either a depth-first or breadth-first traversal. The objects that are reached are marked in some way: either altering bits within the objects or by recording them in a bitmap (or other data structure).



[JL96] The graph after the marking phase. All unmarked cells (with unshaded mark-bits) are garbage.

Figure 2.2.4.1 Mark and Sweep

2. **Sweep Phase** – reclaim the garbage.

Once the live objects have been made distinguishable from the garbage objects, memory is swept, that is, exhaustively examined, to find all unmarked (garbage) objects and reclaim their space.

The typical problem with mark-sweep collection is that the cost of a collection is proportional to the size of the heap, including both live and garbage objects. A fundamental limitation is imposed on any possible improvement in efficiency.

Another aspect that makes this approach different from reference counting is that the straightforward version of the algorithm does not support any form of concurrency. The problem is that the mutator might “alter” references that have been already traced by the collector. The situation that can cause trouble is the following:

Cond1: the mutator creates a reference from a marked object to an unmarked object.

Cond2: this is the only reference (the original reference is destroyed).

The solution is to use either read barriers (this prevents the mutator from seeing an unmarked object) or write barriers (all the “dangerous” cases are recorded so that the collector can (re)visit the nodes in question).

2.2.3.3 Copying Collection

The idea is to divide the heap space into two contiguous semispaces (called *FromSpace* and *ToSpace*). The algorithm starts by flipping the semispaces (*FromSpace* becames *ToSpace* and vice-versa). Then each object in *FromSpace* is copied into

ToSpace, along with all its descendants. In order to avoid multiple copying of the same objects that can be reached my multiple paths, a forwarding pointer is installed in the old version of the object. When the scanning process finds a pointer into *FromSpace* the object it refers to is checked for a forwarding pointer. If it has one, it has already been moved to *ToSpace*, so the pointer it has been reached by is simply updated to its new location.

All live objects will be eventually copied into *ToSpace* and the nice side effect of this approach is that it solves the memory fragmentation problem in a natural way.

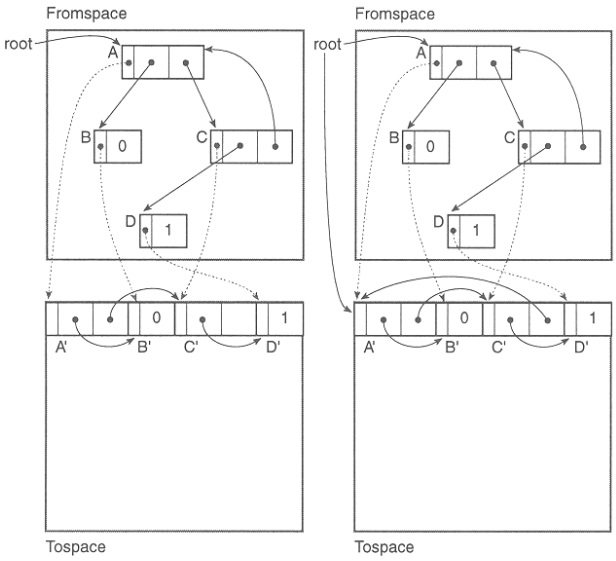


Figure 2.2.3.3 Copying Collection

The main drawback of copying collection is that it halves the effective amount of available storage. Also, this is a stop-and-collect approach (although there are a few incremental approaches written specifically for the ML language).

2.2.3.4 Generational Garbage Collection

Generational Collection is based on the observation that some objects live for a shorter period of time than others, so it might be a good idea to collect the region of the heap where these objects are located, rather than the whole heap. To the best of our knowledge, this technique does not have a corresponding technique in the distributed garbage collection field, so we are not going to present it.

2.2.4 Distributed Garbage Collection

2.2.4.1 Why is Distributed Garbage Collection Different?

The main differences between non-distributed and distributed garbage collectors are caused by the main difference between local and distributed systems: the presence of message passing mechanism in distributed systems.

Message-based communication (over potentially unreliable connections) results in the following issues, which are specific to distributed collectors:

- How to avoid race conditions? The order of specific events is sometimes important. It might be important that two consecutive requests sent by a node to two different nodes be processed in the same order in which they were issued. Note that this is a problem that will occur even if the communication is error-free. A sample scenario showing how race condition can occur (affecting the safety property, therefore the correctness of the collector) is presented later in this section.
- A new type of overhead – message passing. The message passing mechanism introduces a new type of overhead – the communication overhead, typical to distributed collectors.
- Scalability. Any distributed collector should scale as the number of nodes in the system increases.
- Fault tolerance – robustness against message delay, loss or replication, or process (node) failure.

- Collaboration local - distributed collector. Some distributed garbage collectors consist of slightly modified versions of uniprocessor collectors (one per host) combined with one inter-space collector. Ideally, the two types of collectors should be as decoupled as possible, since this would allow custom inter-space space collectors to be implemented.
- Decoupling local GC – mutator. Some research [P96] suggested that in environments that support object migration it is bad to use distributed collectors that rely on the structures also used by the mutator (the object finder or the forwarding pointers).
- Distributed cycles.

All issues have to be considered, but the first one mentioned above is particularly important since the correctness of the distributed algorithm depends on it.

The following scenario shows how a race condition can occur in a distributed garbage collection algorithm based on naïve reference counting.

1. Host A sends a copy of the object P (owned by Host C) to host B
2. A destroys its reference to P and notifies C
3. C (the owner of P) receives the message, and thinks that the last reference to P has been deleted, and therefore destroys P
4. B receives the copy of P from A, and notifies C
5. C realizes that P was incorrectly deleted! (or, even worse, will think that a new reference has been created to some other object Q that took P's place)

Figure 2.2.4.1 illustrates the scenario described above (R represents the root node on each machine).

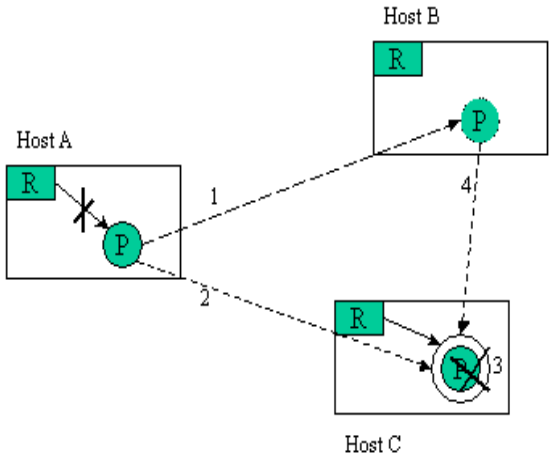


Figure 2.2.4.1 Race condition in distributed garbage collection

Similar problems will occur in mark-and-sweep (tracing based) algorithms, where there is a need for synchronization between the individual mark phase and the distributed sweep phase. Reference counting based, mark-and-sweep and other types of distributed garbage collected algorithms are discussed below.

In the remainder of this section we will discuss some approaches for distributed garbage collection which were were either relevant to our work with JSDA (network objects, Java RMI) or just classical approaches (weighted reference counting, Hughes' algorithm). We are aware that there are approaches for distributed garbage collection other than those discussed here, but, to the best of our knowledge, we did not omit any well-known technique or any algorithm that could be suited to the JSDA framework.

2.2.4.2 Distributed Reference Counting Collectors

Distributed reference counting is a simple extension of uniprocessor reference counting. There is counter associated with each object, but it is usually used in a slightly different way. Each object stores in its associated counter a value representing the number of hosts (not objects!) holding references to it. When the counter value drops to zero, the object is no longer remotely referenced and the problem is therefore reduced to the uniprocessor case. Just like in the non-distributed case, the benefit of being able to interleave small steps of the collection with the computation is preserved.

A new problem that has to be solved in the distributed architecture is the prevention of object reclamation while references to it still exist. This may happen if messages arrive in an order different from that expected. For instance, if a message deleting the last reference to an object overtakes a copying message (the copy of a reference to another node), the object will be reclaimed incorrectly. One way of dealing with this is described later in this subsection.

Another problem with this approach is the way it deals with duplicate messages: this will result in premature collection of objects. However, using reference listing can solve this.

A third remark has to be made here: this type of collector is considered to be better suited to loosely-coupled architectures, since any reference copy/deletion involves a control message (increment/decrement) sent to owner of the object. There are uniprocessor techniques (deferred reference counting) that do not send a message every time. [DB76] describes such an algorithm (implemented later in Smalltalk), which treats

local variables and stack allocated temporaries differently – no reference count bookkeeping is done when they are modified. However, it is not clear whether this kind algorithm can be extended to distributed environments. On the other hand, it is also not clear at all that tracing based algorithms perform better from this point of view (amount of communication overhead), since tracing is based on the examination of *all* objects and therefore a lot of messages are sent. It is hard to say which approach scales better in general, and I believe that each algorithm needs to be evaluated separately. Another reason that makes this comparison hard is that only a few of the surveyed algorithms have been implemented, and there was no work at all in trying to compare them. This is primarily because each approach is tailored to a specific system.

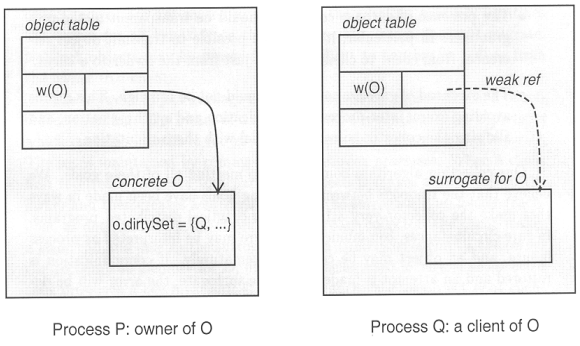
Network Objects

Birell ([BEN+93]) proposes a distributed reference listing collection algorithm to support distributed object-oriented programming. The authors present an outline of their method but do not give implementation details. Our distributed garbage collection algorithm (Section 3.2) is a variant of their method.

Objects visible to other nodes are called *network objects*. Client processes may hold references to the *concrete* object (the "real" object) through a *surrogate* object that communicates with the owner through remote procedure calls. The owner is the host which contains the concrete object and it also contains the list of references to each object (stored inside of the object, in a structure called *dirtySet*).

I am not going to discuss the approach in detail, but mention instead the new ideas it brings and the mechanisms that it uses for race condition avoidance and fault-tolerance.

A new concept is *weak reference*. A weak reference is a special kind of reference which (unlike a normal reference) allows the referred object to be eligible for garbage collection. This concept was introduced later in Java at the API level in order to provide some limited interaction with the local garbage collector. One important benefit is that the local and global garbage collectors are decoupled, and any type of local collector can be used as long as it provides the weak reference mechanism. Figure 2.2.4.2 shows how the surrogate/concrete object is referenced in the client and in the server (w(o) represents a unique identifier for object O in the distributed system).



Source: [BEN+93] Object tables at owner and client processes

Figure 2.2.4.2: Network Objects

The possible race conditions are avoided using an acknowledgments-based mechanism. The idea is to keep a reference to the object (either to the surrogate or to keep a dirty entry in *dirtySet*) until the reference transmission is acknowledged. The message overhead consists of only one extra-message, since network objects are transmitted as arguments/results of remote procedure invocations. There is some CPU overhead though.

A good degree of fault tolerance is achieved using diverse mechanisms. All types of message failures are handled: lost messages, late messages (sequence numbers are used), duplicate messages (this is a intrinsic property of reference-listing based algorithms). Process failure (or termination) is detected by sending ping messages.

This approach does not give a solution for the collection of distributed cycles.

Weighted Reference Counting

Weighted Reference Counting (WRC) [B87] eliminates the increment messages. Besides the benefit of reducing the communication overhead, this approach also eliminates the potential race conditions (however, it has its own shortcomings that I will discuss later).

The key idea is that each object has a weight (a large value) that is split when the object is copied to another host. A part of the weight is sent along with the copy of the object. Decrement messages are sent when remote references are collected and are also accompanied by the former reference's weight.

The problem with this approach is that after a reference is copied for a certain number of times, the associated weight cannot be further divided (it will become 1). Some improvements have been proposed ([G89]) to solve this problem; indirection cells are created – these are proxies for the object, with their own tree of references.

However, there is still a big problem that has to do with the nature of this approach: every message sent is "important", because if a part of the weight is lost then there will be no way to recreate the weight and consequently the corresponding object will never be garbage collected (the liveness condition is broken). Since there is no additional mechanism to ensure message failure handling, this approach is not fault-tolerant.

WRC does not collect cycles.

Garbage Collection in Java RMI

Java RMI (Remote Method Invocation – [Rmi98]) uses an interesting mechanism: a reference to a remote object is *leased* for a period of time by the client holding the reference. This means it is the client's responsibility to renew the lease – by sending dirty() messages - until it expires.

It is interesting to take a quick look at the DGC interface in the java.rmi package:

```
public interface DGC extends java.rmi.Remote {

       Lease dirty(ObjID[] ids, long sequenceNum, Lease lease)
              throws java.rmi.RemoteException;
```

```
        Void clean(ObjID[] ids, long seqNum, VMID, boolean strong)
                throws java.rmi.RemoteException;
}
```

The `Lease` object contains `VMID` (a virtual machine identifier – a unique identifier for the process who own the object) and a `duration` after which the lease expires unless renewed by the client.

RMI uses a reference-listing mechanism. However, the clients do not need to send messages to the RMI objects every time they create/delete a reference to it, because the validity of the lease implicitly states that the client holds a reference.

There are a few characteristics of the RMI collector that are determined by the use of leases (a plus sign denotes a strength while a minus denotes a weakness):

- It is conservative
- It has a relatively small message overhead (+)
- It is fault-tolerant (+)
- Cannot collect cycles (-). The RMI specification does not discuss distributed cycles collection, and I believe that this problem is not solved (unless some additional mechanism which was omitted from the specification is used).
- Race conditions ?

  It seems that RMI does not provide a way for avoiding race conditions in the garbage collection algorithm. However, race condition situations are not very likely to occur in practice because of the client-server architecture promoted by RMI. A possible scenario in which RMI will probably fail is the following:

1. Client A sends B a reference P to an RMI object to another client
2. A destroys the reference to P
3. The lease expires and the RMI server collects the object
4. B receives the message from A and tries to refer a collected object

2.2.4.3 Distributed Mark-Sweep (Tracing-Based) Collectors

The standard approach is to combine independent, per-space collectors, with a global inter-space collector. The two types of collectors interface to each other through exit items and entry items.

The mark phase is complete when all the reachable objects have been marked and there are neither marking nor acknowledgement messages in transit. Afterwards, each space triggers independently a sweep phase on order to reclaim garbage objects.

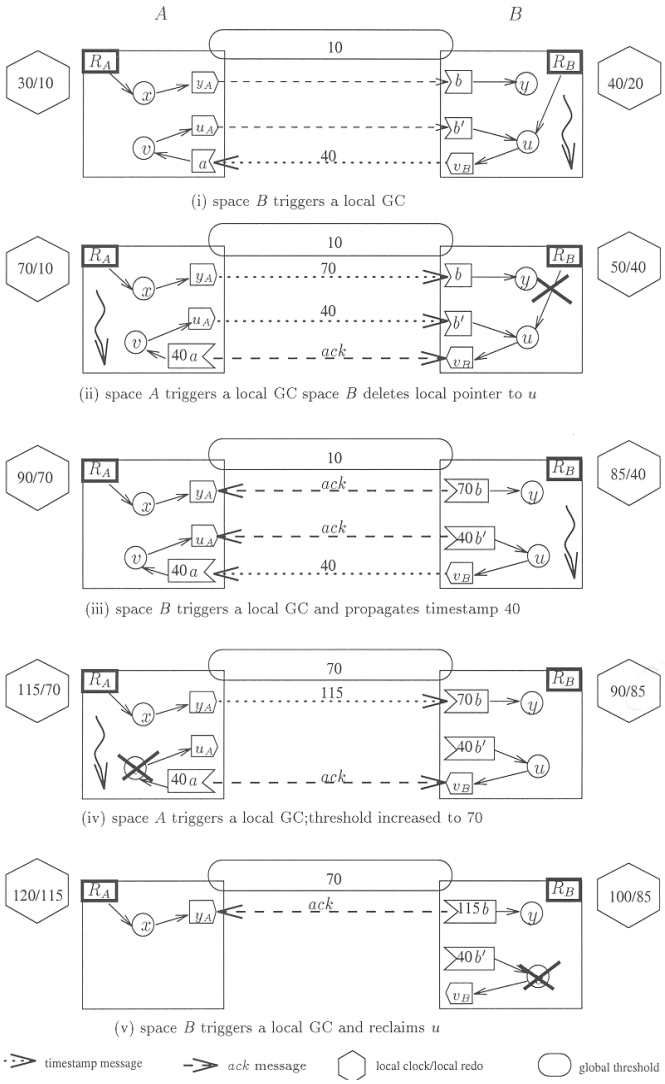
Obviously, one of the problems with this approach is caused by messages in transit. Another issue is fault tolerance.

Tracing with Timestamps

One of the first distributed mark-and-sweep algorithms [H85] used **timestamps** instead of mark bits.

The key idea is that a garbage object's timestamp remains constant whereas a non-garbage object's timestamp increases monotonically. It is safe to say that any entry item whose timestamp is lower than a global threshold is garbage.

Each local GC traces objects from the local root and from the entry items. An item reachable from the local root is marked with the GC-time, one reachable from an entry item receives that item's timestamp.



(i) space B triggers a local GC

(ii) space A triggers a local GC space B deletes local pointer to u

(iii) space B triggers a local GC and propagates timestamp 40

(iv) space A triggers a local GC;threshold increased to 70

(v) space B triggers a local GC and reclaims u

Source: [PS95]

Figure 2.2.4.3A: Hughes' Algorithm

The threshold is equal to the lowest value of all *redo*s (each space maintains a local *redo* timestamp equal to the greatest timestamp propagated).

One problem is that the threshold computation relies on a termination algorithm, which is notoriously costly and not scalable.

Moreover, the algorithm is not resilient to space failures. One good property (a natural property of all tracing-based algorithms) is the ability to reclaim cycles of garbage.

Acknowledgements are sent in order to preserve the safety property (to avoid race conditions).

The algorithm is not tolerant to space failures.

Even a “slow space” (a space running a conservative collector) will seriously affect the performance of such a collector. If one space does not propagate its timestamps, this will prevent the update of the global threshold and therefore the distributed collection will be stuck. This is true even if the slow space does not hold any remote reference.

Tracing within Groups

A different and more scalable approach ([LQP92]) is to do **tracing within groups**. A group is a dynamic collection of spaces that may overlap or include other groups. The dynamic property of groups allows the removal of failed spaces (in order to not block garbage collection).

The algorithm proceeds in several steps: group negotiation, initial marking (distinguishes inter-group from intra-group references), followed by a global marking (performs mark-and-sweep within the group).

Local marking (propagation):

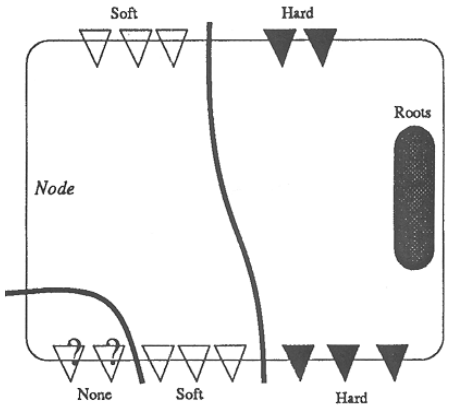
- initially, all marks on exit items are reset to *None*

- a first tracing is done - from hard entry items and roots

- a second tracing is performed starting from entry soft items (exit items that are reached are marked *soft* only if they have not already been marked *hard*)

- *None* items can be reclaimed

Figure 2.2.4.3B below shows the colors of the items after the local marking.



Source: [Lan92] Two-phase marking in a local GC

Figure 2.2.4.3B: Tracing within groups

Global marking (propagation):

Hard marks are propagated to the referenced entry item, whenever it belongs to the group under consideration. This phase ends when the **group stability** condition holds (this requires that no more hardening is possible). Essentially, this is a distributed termination problem, therefore it will add some additional overhead to the system.

At the end, soft entry items belong to distributed cycles! They can be reclaimed.

This approach cannot deal with message failures.

If a node fails to cooperate, the group it belongs is reorganized to exclude the node, and collection continues.

2.2.4.4 Object Migration

An interesting approach that does not fall in any of the categories mentioned above is object migration ([Bi77]). The basic idea is very simple: instead of sending messages among hosts, try to migrate objects in such way that distributed collection (distributed cycles collection) would not be necessary. In other others, transform distributed cycles in local cycles that can be removed by any tracing-based local collector.

A key question is how to choose a good heuristic in deciding which object is suspected to be garbage and should be migrated.

However, there is a more serious problem with this approach – it does not accommodate indirections well. This means that is possible that migrating an object might result in the creation of new surrogate objects, complicating the reference chains.

Another aspect is that objects should be normally moved based on criteria like load balancing rather then garbage collection. The garbage collector should interfere with the mutator as little as possible.

2.2.4.5 Another Taxonomy and a "new" type of DGC

The classical taxonomy is to divide the DGC algorithms in: reference counting, mark-sweep and maybe, hybrid approaches (use distributed reference counting and some additional technique – e.g.: a mark-sweep collector or object migration – to reclaim distributed cyclic structures).

Jones [J00] proposes a new taxonomy: the DGC algorithms are divided in: indirect and direct GC, and each category has the subcategories: tracing and non-tracing.

The approaches mentioned earlier in this paper fall into this taxonomy as following:

1. Indirect, non-tracing GC – [Rmi98]
2. Indirect, tracing GC – [LQP92] (partitioned), [H85] (autonomous)
3. Direct, non-tracing – [BEN+93]
4. Direct, tracing – ?

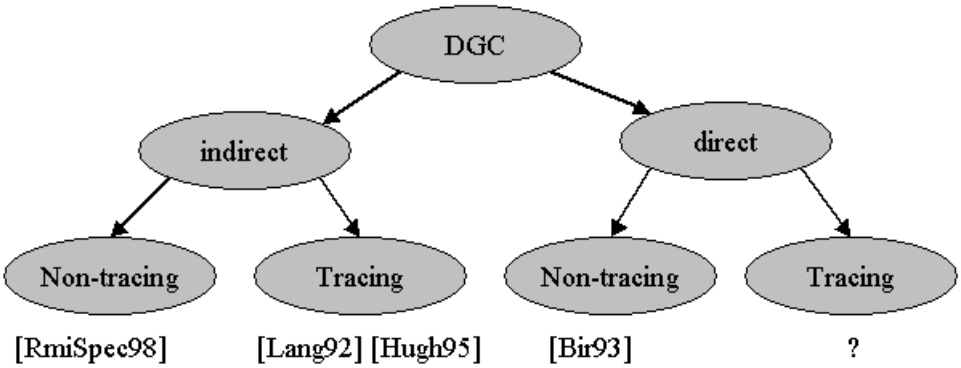


Figure 2.2.4.5i The Taxonomy Proposed by Jones and Lines

The purpose of this taxonomy is to highlight a category that potentially has the benefits of tracing (completeness) and direct algorithms (scalability). This class of algorithms is new (compared to the others). The following two approaches both represent reference counting (listing) algorithms augmented with tracing techniques whose role is to collect distributed cycles.

Back-tracing

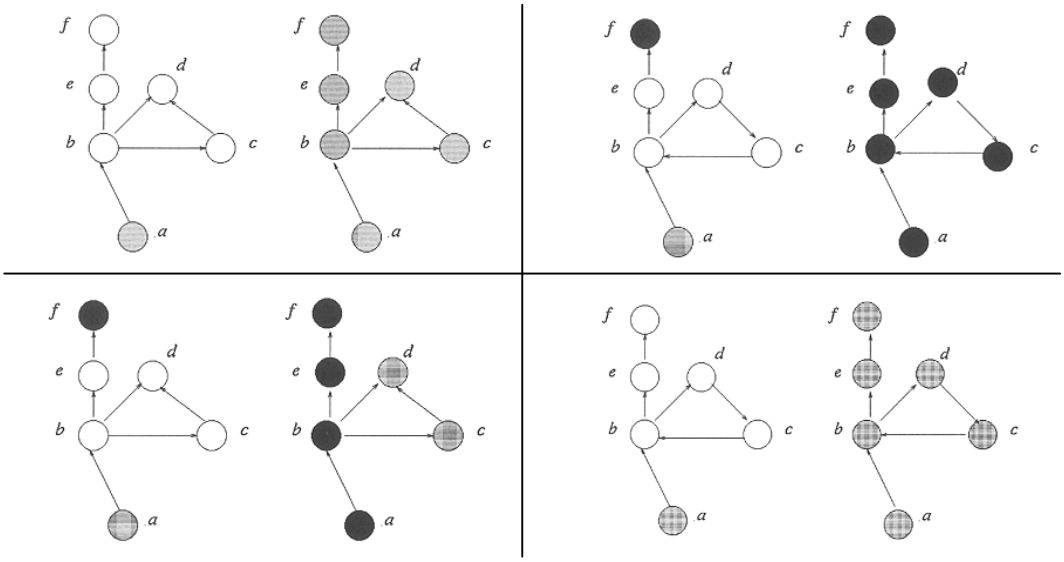
One of the early papers that propose a **back-tracing** technique belongs to Fuchs [F95]. It can be seen as an extension of [BEN+93] – it augments reference counting by maintaining the **distributed inverse reference graph** (IRG): each object maintains a list of pointers to other spaces known to have references to it. The author is trying to show that there is enough information in the IRG to collect cycles with a reasonable overhead.

The key idea is that if we start (back-)tracing from any object and we will encounter a root at some point, then the object is alive. The root nodes are called PR (Persistent Roots) to distinguish them from the Garbage Collection Roots – GCR.

Here are rules for the IRG traversal:

1. Initially all PRs are black, the GCR is grey, and all other nodes are white.
2. Grey tail, white head => grey head
3. Black tail, white head => grey head
4. Grey tail, black head => black tail
5. If none of 1.-4. can be applied to any edge, any remaining grey objects can be collected

Figure 2.2.4.5 depicts the coloring mechanism. In each case of the four scenarios, the graph on the left shows the initial marking and the one on the right the final marking.



Adapted from [Fuch95]

Figure 2.2.4.5 Back Tracing

It looks like we are doing a tracing operation here. However, there are two key-aspects that need to be considered:

1. In a tracing-based algorithm, *all* live nodes are traced! Here, the roots of the tracing are chosen;

2. This approach is incremental.

The collector has to identify the suspects – objects that are likely to be garbage. Fuch does not provide details on how this is done. This would not be a big problem if the algorithm supported a high degree of concurrency.

Unfortunately, overlapping traces cause problems. A primitive algorithm that synchronizes several distributed collectors based on their ID-s (a unique number that determines the priority) is presented, but it seems to incur big overheads.

A limited degree of fault tolerance is provided, since the algorithm assumes that every message is received at least once.

Partial Tracing

[R98] proposes a mechanism for augmenting the network objects model with distributed cycles collection.

Like in back-tracing, only a partial-tracing is done and the roots of tracing are, again, chosen among "suspects". No reasonable mechanism for choosing the suspects is described (the authors mentioned they chose as suspect any object that is not locally referenced).

The algorithm operates in three phases:

1. **Mark-red** – Identifies a distributed subgraph that may be garbage: subsequent efforts of the partial trace are confined to this graph alone; in this step, the (possibly incomplete) transitive referential closure of suspect objects is marked **red**; for each object X traced, a RedSet(X) is created. RedSet(X) contains all the sites that contain references to X.
2. **Scan** – determines whether members of this subgraph are actually garbage; this is done by comparing ClientSet(X) to RedSet(X) for each of the objects marked red. ClientSet(X) is the name used by authors for the X's *dirtySet* (in Network Objects

terminology). If, as a result, an object is detected to be live, then it is marked **green**. In a second step, all objects reachable from local roots or from green concrete objects are now repainted **green** by a *local-scan* process.

This seems to be one of the drawbacks of this approach – a separate local tracing routine has to be implemented by the DGC. The idea is that although the local collector might use a tracing-based local collector, the DGC has no access to it because the network objects model is used.

3. **Sweep** – all red objects are part of distributed cycles. Their entries in the ObjectTable are removed and the Network Object system will take care of their collection.

The mark phase does not need to find the complete transitive referential closure of suspect surrogates. Therefore this distributed garbage collector is **conservative** at the (usual) price – efficiency (and scalability). It is – at some degree - similar with tracing within groups (presented earlier) from this point of view.

Besides the problem mentioned at 2., the algorithm has another weakness: both the scan phase and the sweep phase need to use distributed termination algorithms to detect the end of the respective step.

The authors claim that their algorithm is as fault-tolerant as the Network Objects system. However, it is not clear at all whether their messaging system can be entirely built on top of the messaging mechanism (dirty and clean calls) provided by Network Objects.

The approach has a higher degree of concurrency than back-tracing. However, collectors that belong to different groups cannot interfere.

3. Garbage Collection for Java Distributed Objects

As mentioned in Section 2.1, JSDA uses a distributed object model that is based on stubs and skeletons (skeletons are called *master objects* in JSDA) – an approach similar to other popular architectures like CORBA and RMI.

In fact, any distributed object model has to rely on such a mechanism or one that will expose similar functionality – we need some data structures associated with each object, both on the client(s) who use the object and on the server who owns it; these data structures need to be decoupled from the object itself since their functionality is not intrinsically related to the object they serve. As discussed before, the functionality of a stub/skeleton is merely to marshall/unmarshall a remote request involving the object they are associated to. Since they are all used in the same manner (no matter what object they refer to), usually it is possible to generate them automatically. This is what most of frameworks do (CORBA, RMI fall in this category) in order to simplify programming.

We stressed the generality of this mechanism in order to highlight the fact that the distributed garbage collection algorithm that we propose is applicable not only to JSDA. It can be applied to any distributed object model which uses the stub/skeleton paradigm, as long as a local collector is available for the language and at least a limited interaction with the collector is possible (this last requirement will be further detailed in the following sections).

We chose JSDA because it is a framework for distributed applications that employs a distributed object model and we were more than familiar with its design and implementation.

3.1 The Distributed Object Model in JSDA

There are a few notations that will be used whenever a scenario involving distributed objects is presented. Here is the description of these notations:

1. A distributed object is represented at an **abstract level** by a square drawn around the object's name. When we use such a notation we assume we are not aware of the existence of stubs and skeletons. This will be the case when we want to depict relationships among objects at a high-level. For instance, the user of JSDA (the programmer) will probably think at this level while designing applications. If he/she wants to – let's say – create one object on host X, he will write his code like this

```
/* [ on host X ] */  MyObject A = new MyObject(params);
```

and he/she will know that object A will be created on host X. That is all the user wants and needs to know in regard to where the object resides.

2. At a lower level (inside the JSDA kernel), a JSDA object consists of a master object and several stubs. The master object will be represented by a double circle around its name, whereas a stub will be drawn with just one circle around it.

3. A reference from one object to another will be shown as an arrow.

Figure 3.1 below shows how the links between objects (at an abstract level – the top of the diagram) are represented inside the JSDA kernel (the bottom).

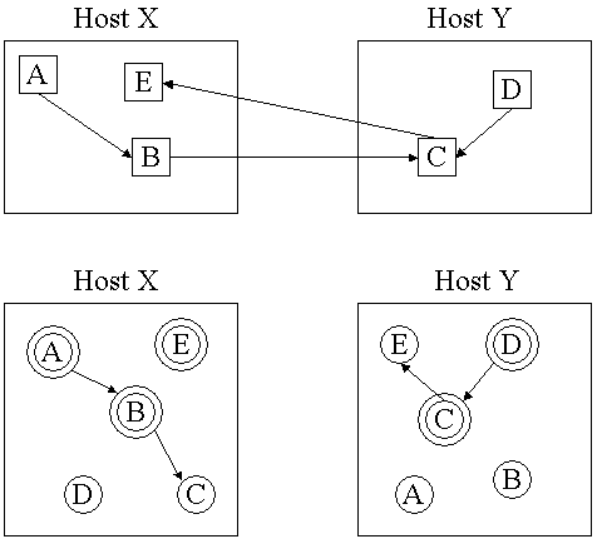


Figure 3.1 Distributed Objects – Abstract model vs. implementation (JSDA)

3.2 The Need for Distributed Garbage Collection. The JSDA Runtime

The bottom part of Figure 3.1 still represents a simplified view of the relationships among objects in JSDA. For instance, the master object for E seems to be unreferenced on host X, although object C holds a reference to it on host Y. This means it could be reclaimed by a local collector. This is not the case though, since there are a lot of **artificial references** to objects in JSDA. By artificial references, we mean references created by the JSDA runtime; these references are totally unrelated to the semantics of the distributed application and the user is not aware of them. Their existence results in a

big problem though – they prevent *any* object from being garbage collected! We will describe now how exactly the artificial references occur in JSDA.

As mentioned in section 2.1, the JSDA framework consists of two main components – a parser and a kernel (runtime). The parser generates distributed versions of all the Java classes required by the user's application (this includes the classes written by the user as well as the required classes that come with the Java platform). The JSDA stubs and masters are instances of these distributed classes. The code generated by the JSDA parser contains invocations of kernel methods. The engine consists of three main modules:

1. The **Engine** – all the kernel invocations must conform to a certain API exposed by the Engine (this API is used internally by the JSDA and its purpose is to decouple the Parser and the Kernel). One of the most important roles of the Engine is to ensure the consistency of the distributed object model or **the unification of address spaces**. In other words, it has to ensure the stub for A on host Y and the master for A on host X refer to the same distributed object. It must keep track of all the existing distributed objects and must maintain – on each host – a JSDA-wide unique identifier for each distributed object. (The master A on host X and the stub for A on host Y both have the same ID.)
2. The **Cache** module – provides caching capabilities.
3. The **DTC** (Distributed Thread Controller) module - provides the entire communication infrastructure along with a mechanism for ensuring that messages only circulate among instances of the same distributed thread. A JSDA distributed thread consists of one thread on each JSDA host.

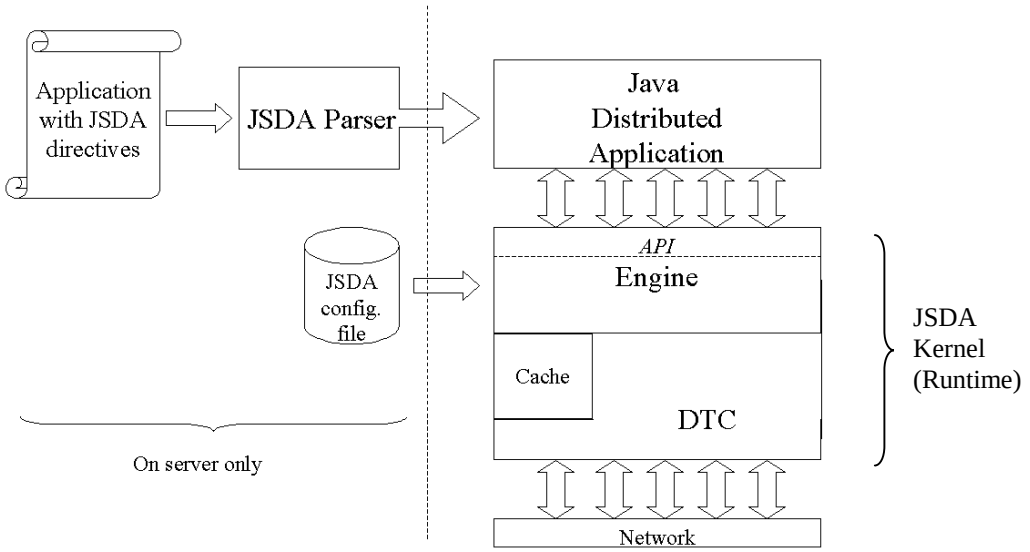


Figure 3.2.1. The JSDA Framework (revisited)

Since the Engine manages the global ID's for each object, it needs to store information so that it can convert from global ID's to local objects (stubs/masters) and vice-versa.

All the invocations coming through the Engine API pass local objects as parameters. If such objects need to be transmitted over the network as parameters of a remote call, the actual values that will be transmitted will be the associated global ID's for each object rather then the objects themselves.

When a remote call is received from another host and passed to the Engine by DTC, the reverse process has to take place – the parameters must be converted from global ID's to their local correspondents. If one parameter represents a stub for an object whose value is needed in the current computation, then an additional remote call will be issue in order to retrieve the primitive data from object's owner.

By storing the correspondence between global and local references we artificially create references to each object in the system and therefore we will prevent them from being garbage collected.

In the old version of JSDA this mappings (local to global and vice versa) are implemented using two hash tables (see Figure 3.2.2 below). However, any type of data structure trying to serve our purpose would generate the same undesired side effect (artificial strong references).

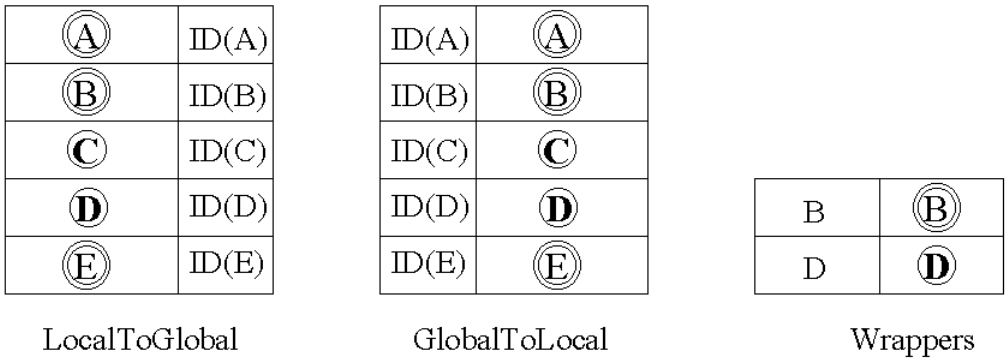


Figure 3.2.2 Artificial references created by the JSDA Runtime on Host X

The third data structure ("Wrappers") in the picture above is necessary in order to deal with the cases when the Engine receives objects that are not in the distributed format (objects whose classes have not been parsed). We are not going to describe here in detail how this can happen in practice, since this aspect is JSDA-specific and it is not related to garbage collection. We just want to mention that this situation can occur as a result of having threads that are not initially controlled by the JSDA runtime – for instance the Java AWT event handling thread (which will generate events as objects). JSDA needs to create on-the-fly distributed versions (called *Wrappers*) of the objects created by these

threads. The Wrappers table stores pairs (object, wrapper) so that the Kernel will not recreate the wrapper for the same object several times. Note that a wrapper is just a JSDA object (either a stub or a master object). In Figure 3.2.2 we just assumed that B and D are objects that need wrappers.

In conclusion, there are data structures inside the JSDA Kernel that – unless modified – will prevent any garbage collector (local or distributed) from reclaiming JSDA objects.

3.3 Interaction with the Local Collector in Java

Any Java platform comes with a built-in garbage collector, since the language does not allow explicit deallocation.

In order to find a solution to the problem described above, we investigated the tools provided by the Java language to allow the programmer to obtain information about the garbage collection process.

3.3.1 Identifying the Available Mechanisms

We looked for any means to discover what the current state of an object is (from the local garbage collector's point of view) and we also tried to identify ways of changing that information ourselves.

Finalization

We started by examining the finalization mechanism provided by Java. The *finalize()* method can be overriden by any Java class in order to provide the programmer with the means of performing any resource deallocation. *Finalize()* is similar at some degree with C++ destructors, one of the differences being that Java finalizers will release resources like sockets and file descriptors rather than memory (since memory management is done automatically).

Java guarantees that if an object has a finalizer, it will be invoked right before the garbage collector decides to collect it. (However, it does not guarantee that an object will *ever* be garbage collected. In that case if the JVM exits, the memory will be freed by the operating system). This allows us to detect the moment of garbage collection for any object. However, this is not enough in order to implement a distributed garbage collector. We will also like to be able to decide not to collect the object (for the simple reason that a remote machine holds a reference to it!) even if the local collector decided the object is not reachable locally. Of course, we don't have this problem in the JSDA, but this is simply because no JSDA object will ever become eligible for collection (which is obviously wrong since the liveness condition is broken).

It is possible to resurrect an object by creating a new pointer to *this* (the current object) inside the *finalize()* method. However, the finalizer will be only executed *once*! Therefore the finalization mechanism cannot provide us with the means to implement a distributed collector.

Reference objects

Before Java platform version 1.2, finalization was the only way to interact with the garbage collector. Version 1.2 came with a small new package (*java.lang.ref*) whose only goal was to provide some hooks to the garbage collector, using **reference objects**.

Sun Microsystem's documentation on Java reference specifies typical scenarios for the use of the classes in this package (described in the next subsection), but distributed garbage collection is not among them. **It was the central goal of our work to determine whether this new feature of the Java language allowed us to build a distributed garbage collector for JSDA and – by extension – for any other distributed object models that uses a similar paradigm.**

On short, a reference object is an object that does not prevent another object referred by it – called the **referent** – from being garbage collected once the local collector determined it is not reachable through normal references. Reference objects are presented in the next section.

3.3.2 Reference Objects

Besides reachable and unreachable objects, the Reference Objects API gives us strengths of reachability. We can have softly, weakly and phantomly reachable objects and gain a limited amount of interaction with the garbage collector according to the strength of reachability.

The Reference Objects API consists of the classes shown in Figure 3.3.2.1 below:

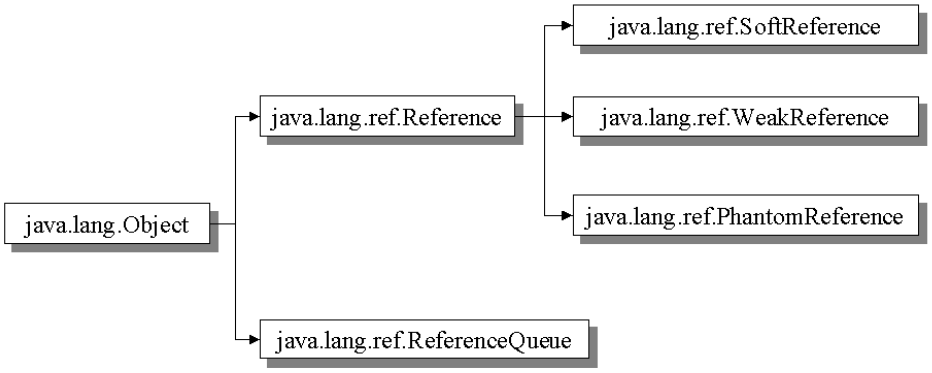


Figure 3.3.2.1. The Class Hierarchy for package `java.lang.ref`

Each reference-object type is implemented by a subclass of the abstract base Reference. A reference object is an instance of one of these subclasses and encapsulates a special kind of reference to another Java object. This reference is called strong, weak or phantom reference depending on the reference object's type (StrongReference, WeakReference or PhantomReference respectively).

From strongest to weakest, the strengths of reachability are the following:

- **Strongly reachable**

  An object is strongly reachable if it can be reached by some thread without traversing any reference objects.

- **Softly reachable**

  An object is softly reachable if it is not strongly reachable and there is a path to it with no weak or phantom references, but one or more soft references

- **Weakly reachable**

An object is weakly reachable if it is neither strongly nor softly reachable and there is a path to it with no phantom references, but one or more weak references.

The *javadoc* documentation from Sun is ambiguous about the real difference between soft and weak references. We discovered that this difference consists of the fact that weakly reachable objects are immediately reclaimed once they have been traced by the local collector, while softly reachable objects survive several garbage collection cycles. More details about weak versus soft references are presented in Section 3.5.

- **Phantomly reachable**

  An object is phantomly reachable when the collector does not find any strong, soft or weak references to it, but at least one path to the object with a phantom reference. Unlike weak/soft reference objects, phantom objects must be registered to a reference queue (described below).

- **Unreachable**

When a (Soft/Weak/Phantom) Reference object is created, two things may be specified:

1. The referent object (this is mandatory)
2. A **ReferenceQueue** object (optional for Soft/WeakReference objects, mandatory for PhantomReference objects)

If a reference object is registered to ReferenceQueue, the local Java collector will place the reference object in this queue when the reference field is cleared (is set to *null*). The reference field is the field that stores the link to the referent.

Reference queues are used to find out when an object becomes softly, weakly or phantomly reachable so the program can take some action based on that knowledge.

The scenario in figure 3.3.2.2 illustrates the use of reference objects.

Step1: a reference object is created

Step2: the reference is no longer strongly reachable

Step3: the reference field is set to null (This happens automatically for Soft/Weak Reference objects – this is the case in our example; Phantom references are not automatically cleared).

Step4: The reference object is added to a reference queue (if is has registered to one).

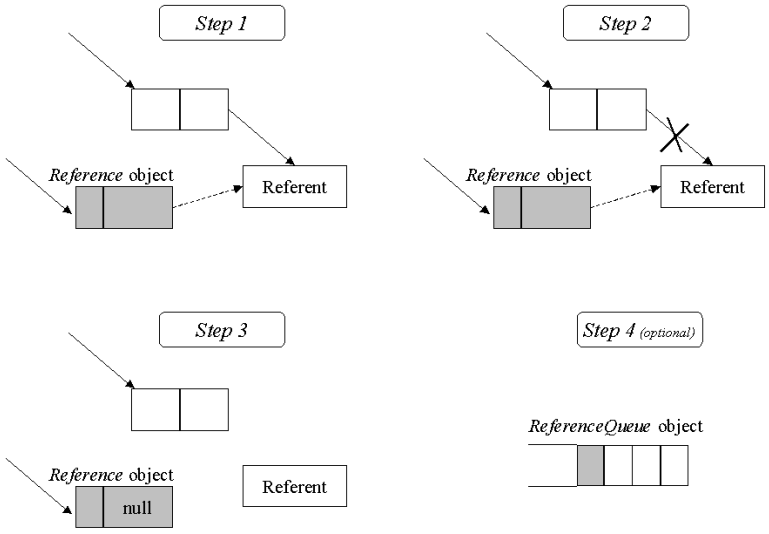


Figure 3.3.2.2. How Reference Objects are Used

Unlike Soft/Weak reference objects, Phantom Reference objects do not have their reference field cleared automatically once they are added to a reference queue. The referent will continue to be phantomly reachable until the reference field is set to null. (This is the reason the API enforces Phantom reference objects to be registered to a queue).

There are typical usage scenarios that the authors if the Reference Objects API had in mind while creating several types of reference objects. They are:

Type of Reference Object	Typical (suggested) usage
SoftReference	Implementation of memory-sensitive caches
WeakReference	Implementation of canonicalizing mappings that do not prevent their keys (or values) from being reclaimed
PhantomReference	Scheduling pre-mortem cleanup actions in a more flexible way than is possible with the Java finalization mechanism

Our goal is to use Reference Objects to implement distributed garbage collection.

The following sections describe our proposal for the solution of this problem.

In the rest of this document, we will usually call **weakly reachable** any object which is neither strongly reachable nor unreachable, since the differences among soft – weak – phantom references are not essential to our discussion.

3.4 Our Solution

3.4.1 Key Ideas

As shown in Section 3.2, there are data structures inside the kernel that prevent JSDA objects from being garbage collected

We wanted to come up with a solution – based on the mechanisms introduced by the Reference Objects – to modify/replace these structures so that the objects will have reachability levels that could be controlled by the JSDA runtime.

One key idea is to **use soft references instead of strong references whenever artificial references are created in JSDA**. This rule alone does not solve the problem and by applying it – without additional mechanisms – we would just give away the safety property in return for liveness. For instance, let us consider the scenario in Figure 3.4.1 below (where we only show strong references, and we assume the master objects A and D are strongly reachable from the local roots):

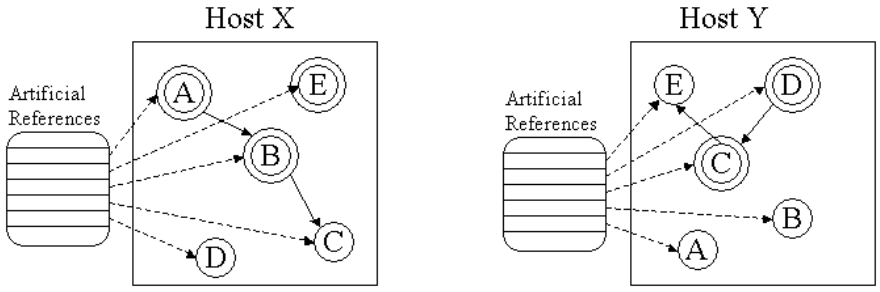


Figure 3.4.1 E and D are unreachable on Host X, although they are reachable on Y!

On host X, the master object E and the stub D will be prematurely collected (if we simply replace the artificial strong references with weak references). Actually, we could recreate the stubs on the fly (see Section 3.5.2), but we cannot do the same for master objects.

In conclusion, we will use reference objects as useful building blocks for some of the new Kernel structures in the distributed garbage collection algorithm, but we will create additional mechanisms to enforce the safety property.

Once we had a solution for eliminating the artificial strong references, we could start thinking about building a distributed garbage collection algorithm.

Since we do not have the possibility of inspecting pointers inside Java objects, we cannot perform any type of tracing; in fact, this is what the Java local collector does, but it is hidden inside the virtual machine and the only API providing information about garbage collection is the Reference Objects API.

Therefore we decided that the only choice was to build a distributed reference counting based collector. Naturally, we chose to create a **distributed reference-listing collector** since they more robust (as discussed in Section 2.2.4.2).

Each master object will have an associated reference list (a bitmap) that will store information about remote reachability for the object. More precisely, there will be one bit for each host in the system, and each host holding a reference to (a stub for) the master object will determine a corresponding bit in the reference list to be set to one. When a stub for an object P is created on a host X, a **dirty** message is sent to the owner of P in order to set the corresponding bit (the bit for X) in X master's reference list. Similarly, a

clean message will be send to P's owner whenever JSDA detects that a local garbage collector collected a stub for P.

3.4.2 New/Adapted Data Structures in the Kernel

In this section we will outline the modifications made to the kernel structures of JSDA that are responsible for artificial references to JSDA objects. We already described in Section 3.2 the role of these structures inside the JSDSA Kernel.

The diagram below (Figure 3.4.2) gives a high-level view of the changes made to these data structures so that they can be used in a distributed garbage collection algorithm. The thick lines delimitate the memory portions that were initially strongly referenced.

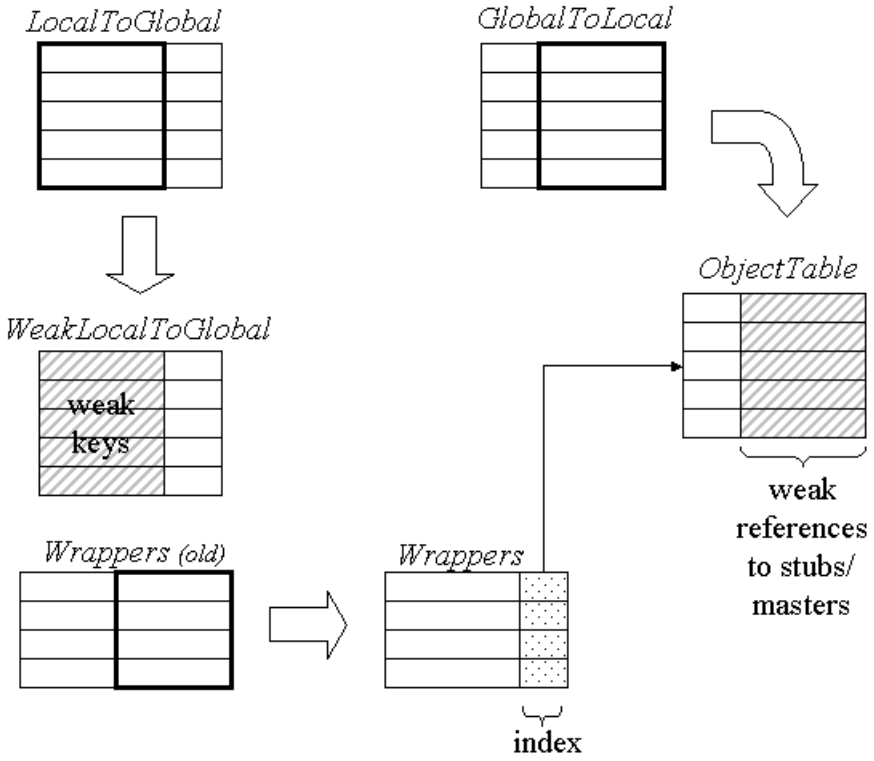


Figure 3.4.2 Modifications of Kernel Data Structures

Here are some details about each of the modified data structures:

1. *LocalToGlobal* was initially the hash table used to obtain the global references (identifiers) for any JSDA object (either stub or master objects). We replaced it with a new hash table whose keys are weakly referenced. As a result, if an object is not reachable in the JSDA application, this artificial reference will not prevent is from being garbage collected.

2. Similarly, the *GlobalToLocal* hash table was used to translate between global and local objects upon receipt of a remote message. We replaced it with a fixed-size array called *ObjectTable*. This object table represents a key point in the new JSDA runtime architecture. It does not hold just the stubs and the master objects for the host it resides on, but also additional information that is used by garbage collection. The following sections give details about this.

3. As described in Section 3.2, the *Wrappers* hash table holds pairs (object, wrapper) for each object that is passed to the kernel in the non-distributed (non-parsed) version. We replaced these occurrences of stubs/master objects by indices that correspond to entries in the object table. These indices represent the object ID-s (that are part of the global ID-s).

3.4.3 The Object Table and the Garbage Collector

The Object Table is shown in Figure 3.4.3.

Any entry in this table can store information about either a stub or a master object. The corresponding Java types are *EntryStub* and *EntryMaster* (which both inherit the abstract class *ObjectTableEntry*). The *type* field of an *ObjectTableEntry* object identifies the type of the entry – it can be either LOCAL (for masters) or NON-LOCAL (for stubs).

The meaning of the rest of the fields is outlined below and further described afterwards:

Stub entries

- object ID – the distributed ID (unique) for the JSDA object
- owner ID – the host ID for the machine who owns the object with the corresponding object ID.
- (Object ID and owner ID together represent the global ID)
- Soft Ref – this field stores a soft reference to the stub
- Strong Ref – this field enables us to store a strong reference to the stub; this reference needs to be set at certain moments, in order to ensure the correctness of the algorithm (in order to avoid race conditions)
- Cnt – counter field; counts the number of on-going remote calls that passed this stub as a parameter; it is used to determine whether the Strong Ref should be set / reset and ensures that the algorithm is thread-safe.

Master entries

- WEAK/NOT WEAK – this field tells us whether the master object is softly or strongly reachable via the Object Table; the JSDA Runtime will decide whether a master object should be accessible via a soft or strong reference in the object table, based on some criteria which are going to be discussed later
- Reference List – represents the reference list (a bitmap) corresponding to the list of hosts that hold references to this stub.

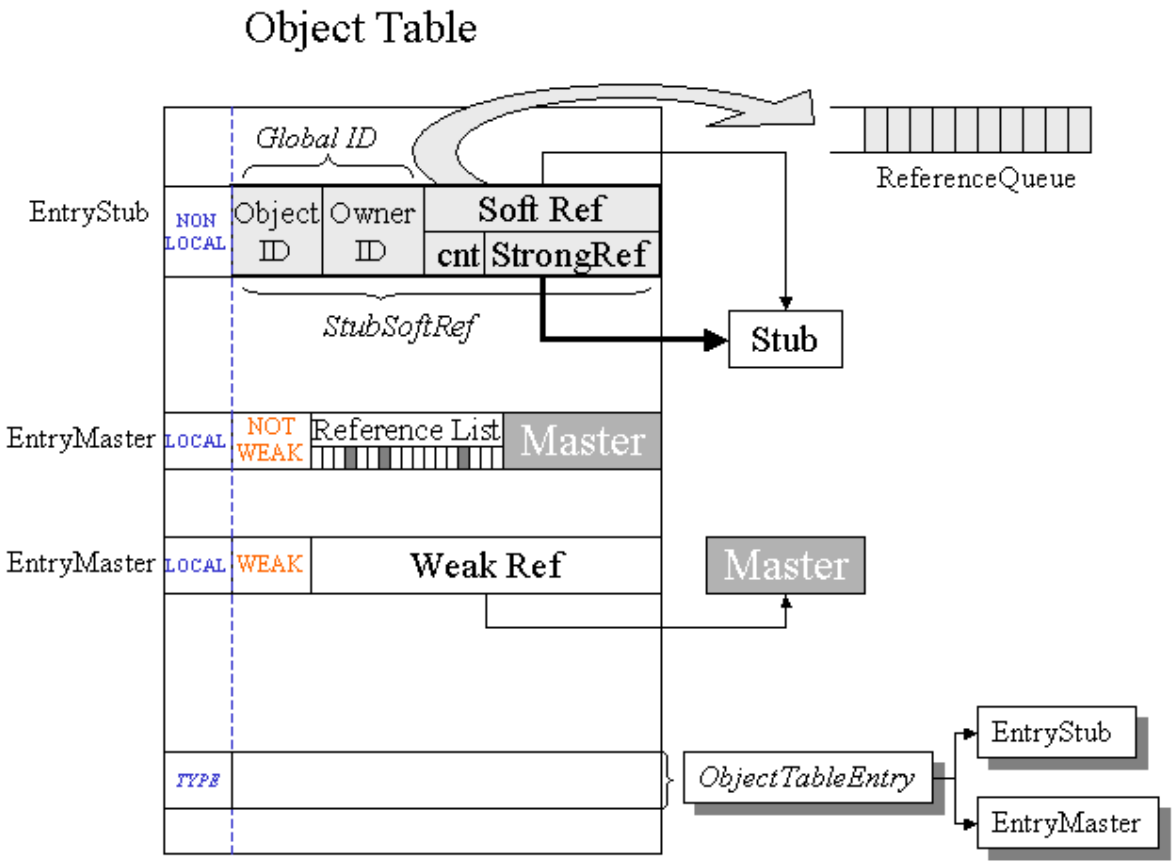


Figure 3.4.3 The Object Table

As explained before, the reference queues are useful in order to allow the programmer to decide on specific actions once weakly referenced objects become eligible for collection. Once a soft/weak reference object is added to a reference queue, the reference field is automatically set to *null*. This implies that there is no way to know which was the referent!

In JSDA, we need to know exactly what stub has been collected since we need to send a *clean* message to its owner. In order to solve this, we decided that the reference objects that we use inside Stub Entries (entries that encapsulate soft references to stubs) must store additional information about the JSDA object.

This is the reason for the redundancy inside Stub Entries in the Object Table. The redundancy consists of the object ID being stored as part of the soft reference object, although it actually represents the index of the current entry in the table. Again, this information is used when the object is removed from the reference queue and we need to establish the JSDA object that it corresponded to.

3.4.4 The Algorithm – Formal Description and Proof

3.4.4.1 Rules

Master Objects

RULE M1 – Master object insertion in Object Table after creation

If the current machine is the JSDA server, then the master object M will be initially stored in the Object Table via a weak reference. Otherwise, it will be stored via a strong reference, and one bit in the *Reference List* field will be set to 1. The bit position corresponds to the *host ID* of the machine who requested the creation on M.

RULE M2 – Dirty calls

If a *dirty* message is received for a master object M:

1. If the entry for M in the Object Table has the WEAK field set, the entry type is changed to LOCAL & NOT_WEAK and a strong reference is created to M
2. The bit inside *Reference List* corresponding to the *host ID* information contained in the *dirty* message is set to 1.

RULE M3 – Clean calls

If a *clean* message is received for a master object M:

1. The bit inside *Reference List* corresponding to the *host ID* information contained in the *dirty* message is set to 0.

2. If the number of bits set to 1 inside the *Reference List* is 0, the entry type is changed to LOCAL & WEAK and a soft reference is created to M

Stubs

RULE S1 – Stub creation

When a process (host) receives a JSDA remote call that contains a global ID with no corresponding local object, the JSDA runtime performs the following actions:

- It creates a stub S on the fly (see Stub Creation in Section 3.5)
- It obtains a global ID from the JSDA server
- It creates a *StubSoftRef* object SR and initializes its fields as following:
  - *Object id* and *owner ID* are set according to the global ID
  - It sets *StrongRef* to *null* and *cnt* to 0.
  - It sets the referent to S and registers the SR object with a reference queue
- It sends a *dirty* message to S's owner and awaits an ACK, if S's owner is not the sender of the remote call.
- It creates a new *StubEntry* object (type is NON-LOCAL) to hold SR, and it stores it in the Object Table at position pointed by the object ID.

DEFINITION 1 – The DGC thread

The DGC thread is a JSDA thread whose sole purpose is to remove the reference objects from the JSDA reference queue and to execute the actions required by garbage collection of JSDA stubs (as specified by RULE S2).

NOTE: only stubs are registered to a reference queue. Master objects are not, since there is no special action that needs to be executed once they are collected.

RULE S2 – Stub collection

When the DGC thread removes a *StubSoftRef* object SR from the queue, the JSDA runtime sends a *clean* message to the host identified by *host ID*, for the object identified by *object ID*. It then removes the entry that stores SR from the Object Table.

Remote Calls

DEFINITION 2 – Reference (object) copy

A reference (an object) – either a stub or a master object – is said to be *copied* to a machine H if it is passed as parameter to a remote call sent to H.

DEFINITION 3 – Reference (object) in transit

A reference (an object) is said to be *in transit* at some moment in time t if $t_{sent} < t < t_{recv}$, where t_{send} and t_{recv} are the moments when the remote call was sent or received by the DTC Kernel module on sender or receiver, respectively.

NOTE: as previously discussed, the JSDA runtime converts local objects to global ID-s before sending them over the network (as parameters of a remote call). Similarly, the receiver obtains the local objects based on the received global ID-s.

RULE R1 – Sending remote calls

When the JSDA runtime receives a request for a remote call to be executed on destination machine D, it performs the following actions for each object O passed as a parameter:

If O is a stub, it sets the *StrongRef* field (in the corresponding stub entry in the Object Table) to point to the stub and it increments the counter field.

If O is a master this is treated as a *dirty* call from D – RULE M2 is applied.

RULE R2 – Receiving remote calls

When a remote call request is received from another JSDA host, each parameter (global ID) is looked up in the *WeakGlobalToLocal* table, in order to find its local instance O.

a) If O is found, no special action is needed.

b) If O is not found, a stub will be created as specified by RULE S1

RULE R3 – Receiving results from remote calls

Upon receiving the result from a remote call, the Kernel will perform the following actions for each stub that was copied:

1. It will decrement the associated counter.

2. If the counter is zero, it will clear the strong reference to the stub from the corresponding Object Table entry (the *StrongRef* field).

3.4.4.2 Proof of Correctness

Liveness

Goal – to prove that all garbage (except for distributed cycles) is eventually collected.

Proof by contradiction: We assume that there is some garbage that is never collected. In other words, the JSDA maintains artificial references to some objects and therefore they cannot be reclaimed. There are 2 cases:

1. An artificial strong reference to a stub is stored by JSDA ⇒ the *StrongRef* field holds a strong reference to the stub. (1)

According to RULE R1 and RULE R3 – which are the only ones that affect the *StrongRef* field – a non-null value for *StrongRef* implies that at least one remote call that copied the object is currently being executed. (2)

(1) & (2) ⇒ there is at least one remote call that is currently manipulating the JSDA object that, consequently, cannot be garbage (contradiction).

2. An artificial strong reference to a master is stored by JSDA ⇒

The master is stored in a NOT-WEAK entry in Object Table ⇒

There is (at least) one bit that is set to 1 in the *ReferenceList* field ⇒

There is (at least) one stub S for this master object that is strongly reachable.

There are 2 cases:

(a) The stub is reachable through an artificial strong reference – this is case 1., which we already proved leads to a contradiction

(b) The stub is reachable through a non-artificial strong reference ⇔ the stub is not garbage - contradiction

Safety

Goal – to prove that no master object will be prematurely collected (remember that we allow stubs to be collected since JSDA has the ability to recreate them).

Proof by contradiction: We assume there are master objects that could be prematurely collected. In order to be *prematurely* collected, a master object M must be collected although either:

(a) It is locally reachable or

(b) There is a stub for it on another machine or

(c) There is reference to it in transit

(a) leads to a contradiction immediately.

(b) implies – according to RULE S1 & RULE M2 – that JSDA has created an artificial strong reference to the master – contradiction

(c) This case can be reduced to either (a) or (b), depending on who is the sender S of the remote call. In both cases RULE R1 is applied.

If S = M's owner ⇒ this case can be reduced to (a)

If S ≠ M's owner ⇒ this case can be reduced to (b)

Basically, the safety property of the algorithm guarantees that no race conditions could occur as a result of message delays in single- or multi-threaded environments.

The next section discusses the safety property from this point of view, presents a few scenarios and shows how race conditions are avoided using the mechanisms presented above.

3.4.5 Race Conditions Avoidance

As shown in Section 2.2.4.1, race conditions can easily occur in distributed garbage collection – resulting in premature reclamation of data – unless special attention is paid to this aspect. We proved that the algorithm that we proposed takes care of this potential problem since the safety condition is ensured. However, we did not fully explain which are the rules that ensure race condition avoidance.

This section presents two examples that show two potential race conditions situations and how the JSDA collector handles them. This should help the reader understand the reason behind using the *StrongRef* and *counter* fields in the Object Table and in the associated rules.

Scenario 1

First, we are going to show how things could work if we did not use explicit mechanisms for race condition avoidance in RULE S1 (the ACK) and RULE R1 (the *StrongRef* field). Consider the following scenario – shown in Figure 3.4.5.1 – involving 3 hosts (X, Y and Z) and one JSDA object that is owned by Z and has a stub on X, accessible from X's root:

I. A remote call is sent by X to Y and initiates the copy of A.

II. The strong reference to A on host X is destroyed by the JSDA application

III. The remote call is received by Y, a stub is created and a *dirty* call is sent to Z. Meanwhile, on host X A is identified as garbage and a *clean* message is sent to Z.

IV. The *clean* message reaches Z before the *dirty* message sent by Y;
Z updates A's reference list and incorrectly determines that A is garbage and reclaims it.

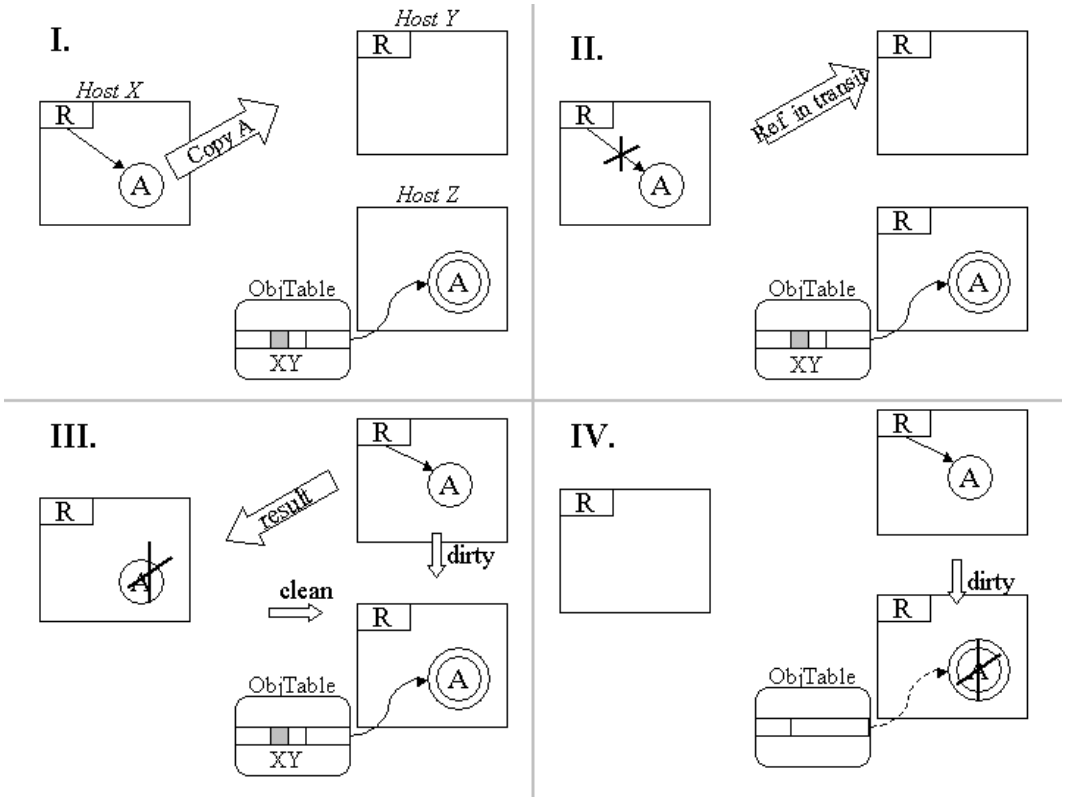


Figure 3.4.5.1 Race Condition – Scenario 1

The solution to this problem is to ask the sender of the remote call to keep artificial strong references to all stubs that are copied to remote machines as a result of the remote call, until the result of the remote call is received. A second condition is that the receiver is required to wait for the owner to receive the *dirty* message before sending the result to the initiator of the remote call. This is the rationale before RULE R1 and RULE S1. Note that RULE R1 also deals with the case of references to masters being copied. In that case, the master is required to set a bit to 1, which will also result in a strong artificial reference being created (if such a reference does not already exist).

Figure 3.4.5.2 shows the correct behavior, implemented by the Kernel:

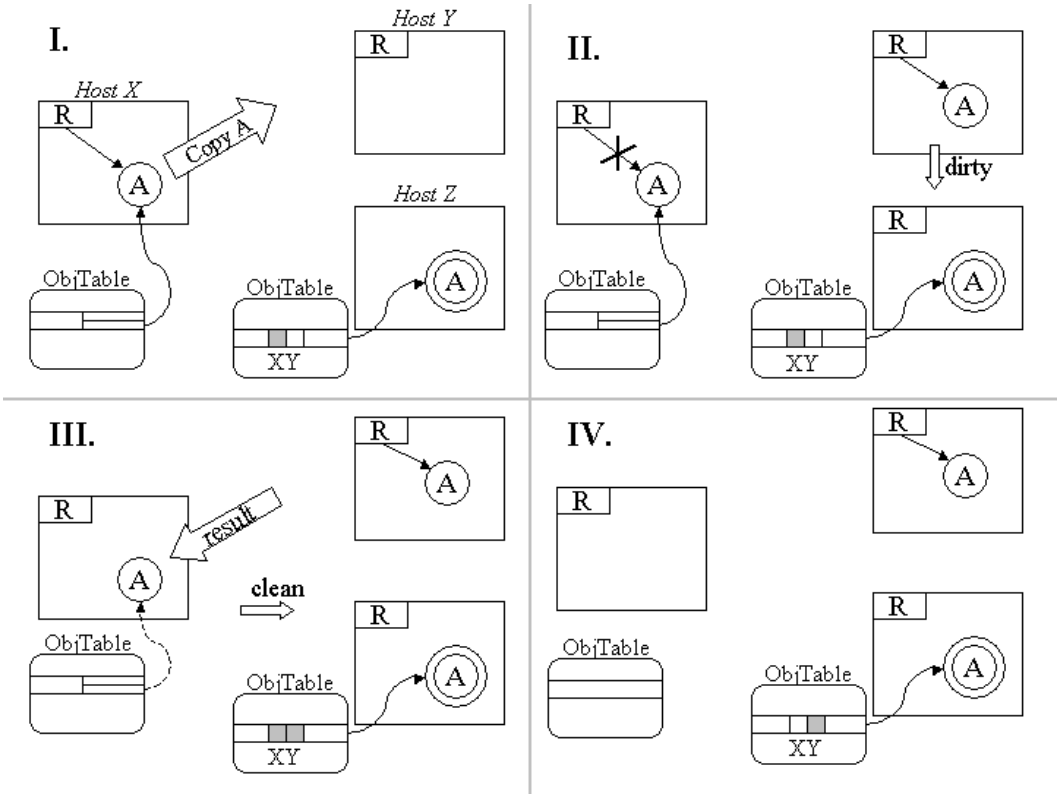


Figure 3.4.5.2 Avoiding Race Condition in Scenario 1

I. The remote call is initiated ⇒ an artificial strong reference is created for A

II. The reference to A on host X is destroyed by the JSDA application

Y creates the stub A and sends a *dirty* message to Z.

III. X receives the result and clears the artificial strong reference. A becomes garbage and a *clean* message is sent to Z.

IV. The master A is not collected since its reference list contains the correct information.

Scenario 2

The mechanisms used above are sufficient in a single-threaded application. The following example will show that the algorithm would not be correct in a multi-threaded environment without the use of the *counter* field inside Stub Entries.

I. The remote call is initiated ⇒ an artificial strong reference is created for A (*StrongRef* is set to A, in A's entry in the Object Table)

   Y receives the remote call request, creates the stub and sends a *dirty* message to W

II. The reference to A on host X is destroyed by the JSDA application. Another remote call is initiated (by a different thread) – destination host is Z ⇒ *StrongRef* is set to A again, in the Object Table

III. Z receives the remote call request, creates the stub and sends a *dirty* message to W.

   Y receives the ACK from A's owner (W) and sends the result of the remote call to X

IV. X receives the result from Y and clears the artificial strong reference ⇒ A becomes unreachable therefore a *clean* message is sent to W

   Meanwhile, the reference to A on host X is destroyed by the JSDA application ⇒ A becomes garbage on Y and a *clean* message is sent to W

PROBLEM: if the *clean* messages from both X and Y reach W before the *dirty* message from Z, W will incorrectly decide that the master object A is garbage!

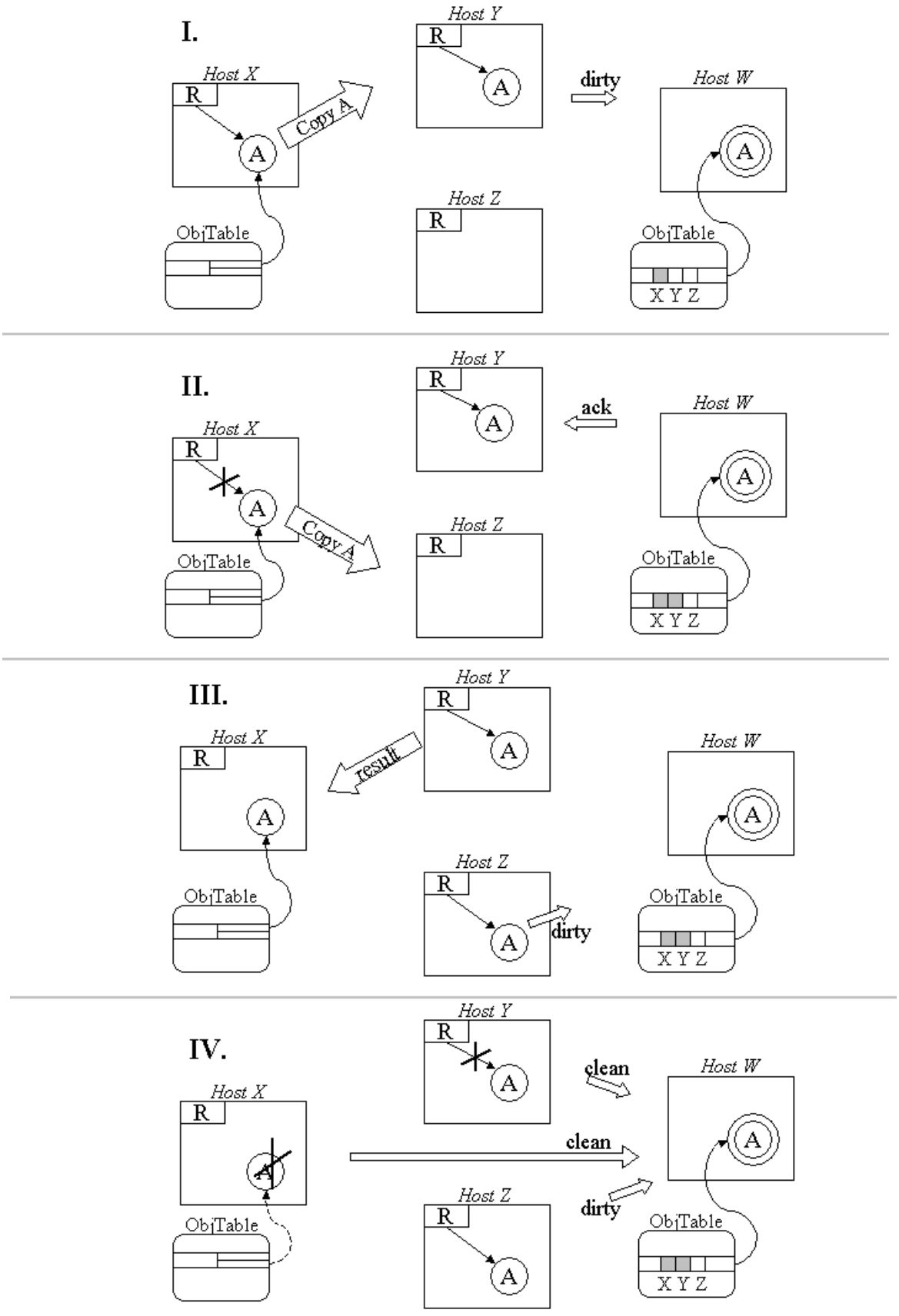


Figure 3.4.5.3 Race Condition - Scenario 2

Conclusion: in a multithreaded environment, it is not enough to keep artificial strong references to all the stubs being copied until the remote call returns. The problem is that a returning remote call will clear the *StrongRef* although another thread copying the same stub(s) might be executing.

Our solution was to maintain a counter (the *cnt* field in the *StubEntry*) for each stub, and to increment or decrement the counter every time a remote call copying the stub is initiated or returns, respectively. The *StrongRef* field is set to *null* upon returning from a remote call only if the counter is zero (there are zero threads copying that stub). This policy is included in RULE R1 and RULE R3.

By applying these rules, step IV. above will change since the result from Y will decrement the counter to 1 and the *StrongRef* field will not be set to *null*. The artificial strong reference will only be destroyed when the result from Y will be received, as shown in Figure 3.4.5.4 below.

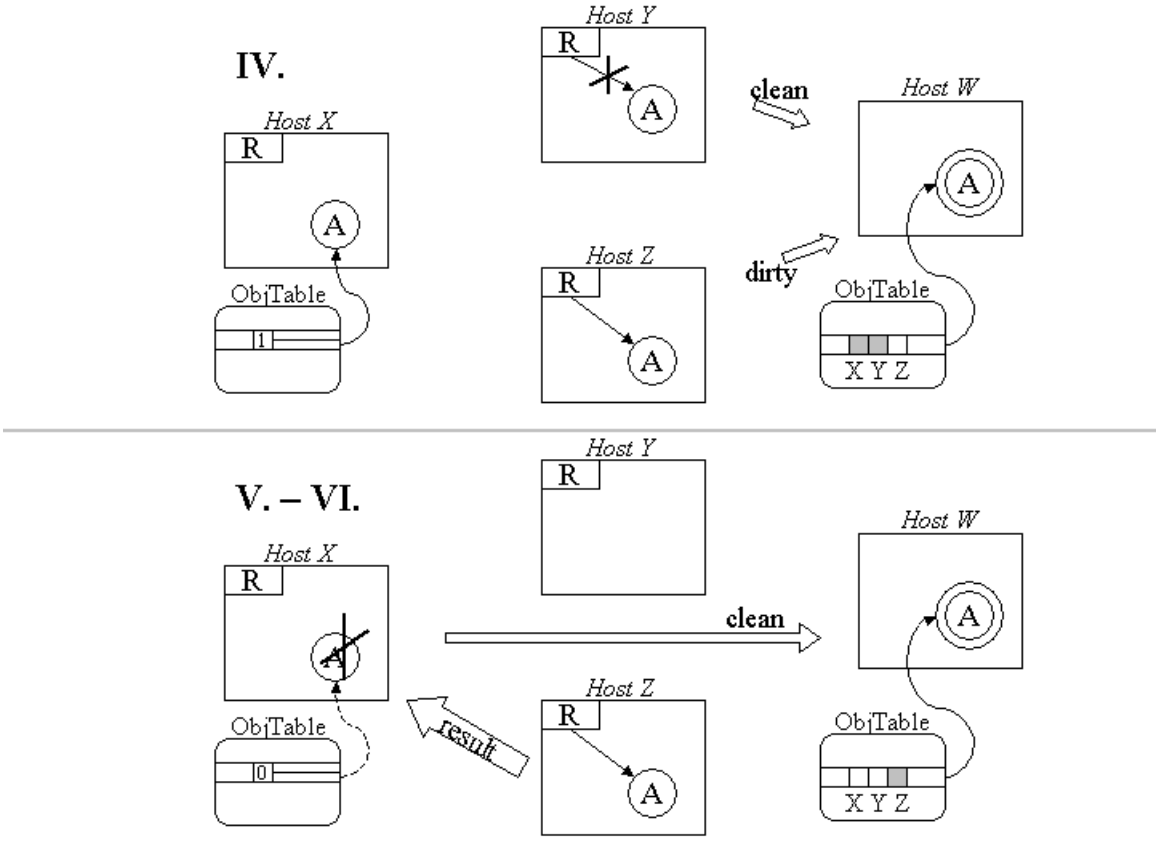


Figure 3.4.5.4 Making the Algorithm Thread-Safe

3.5 Development and Implementation

3.5.1 The Development Approach

We wanted to decouple all the changes and additions that were necessary to make to JSDA, so that that the development process could be an incremental one, allowing us to test new features as we implement them (rather than after the whole algorithm has been implemented). We also tried to keep – as much as possible – a clear separation between the separate Kernel modules. The new Distributed Garbage Collection (DGC) module comprises most of the functionality that is specific to the distributed collector, and it is implemented in a separate package – *jsda.kernel.dgc*.

The initial approach was to create a separate Engine API that we could use in order to be able to test various scenarios involving distributed garbage collection. By doing this, we could skip the parsing step during testing and therefore speedup the development process. We developed and tested this API in parallel with some Kernel improvements (like on-the-fly stub creation). Then we merged the two versions and built the DGC module. In the end, we tested the final version with simulated applications – using the simplified API – first and then with real applications.

The development process can be summarized as following (this represents – roughly – the approach that we followed):

Phase 1:

A. Develop an Engine API to be used for simulating JSDA applications in order to test garbage collection

B. Make changes to the Kernel, which can be tested prior to the full implementation of the distributed garbage collection algorithm.

- New global ID's & on-the-fly stub creation
- Object Table (limited functionality could be tested at this stage)

Phase 2: Merge the versions resulting from Phase 1

Phase 3: Implement the DGC algorithm

- Clean and Control threads (DGC threads described in 3.5.2.3)
- The functionality associated with the Object Table which was note implemented in Phase 1B

Phase 4: Test using:

A. Different scenarios using the modified API

B. JSDA applications

Figure 3.5.1 illustrates the development process.

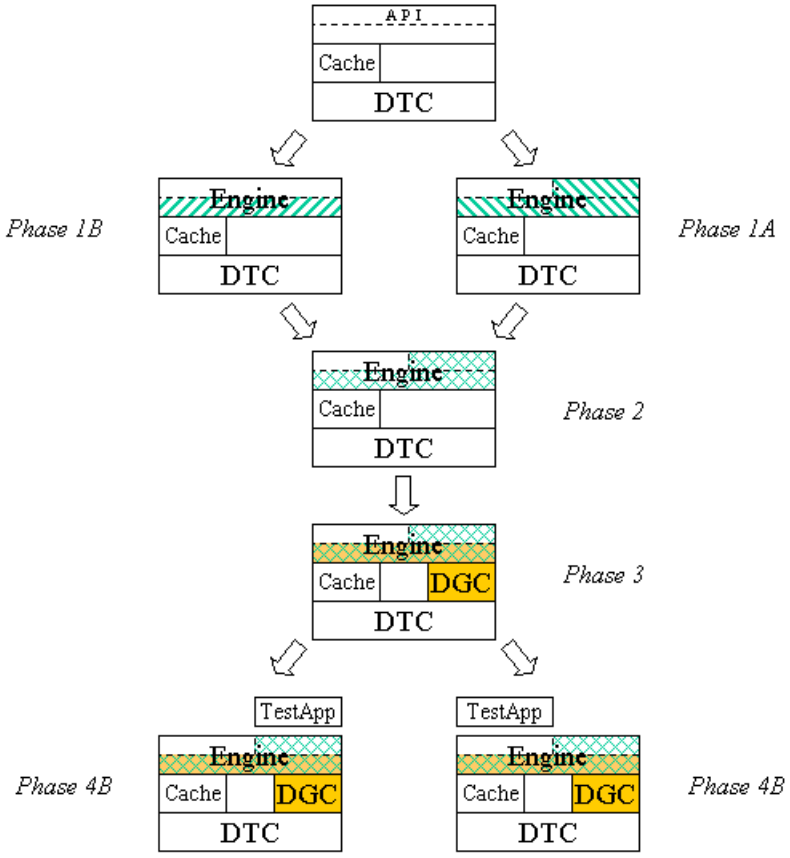


Figure 3.5.1 The Development Process

3.5.2 Implementation Aspects

This section explains important decisions that we made during the development process. They were mentioned earlier, but we considered them lower level aspects so that we postponed their description until now.

On-the-fly stub creation is one of the main features implemented in Phase 1B. The distributed collector needs this Kernel capability since our algorithm allows stubs to be collected in order to ensure liveness.

This section also discusses the differences between weak and strong references, and explains why we decided to use soft references in some places and weak references in others.

The last aspect covered deals with the threads inside the DGC module (*Phase 3*), and shows how they relate to the other – application and Kernel – threads in JSDA. We describe what we had to do in order to keep the JSDA thread model consistent.

3.5.2.1 On-the-fly Stub Creation

As mentioned in Section 3.4, we must not keep permanent strong references to stubs, (or otherwise we break the liveness property of the algorithm). Because of this, the Kernel needs to be able to (re)create stubs on-the-fly.

In the initial version of JSDA, the stub creation was part of the distributed object creation – when a master object was created, stubs were also created on *all* JSDA hosts.

The class name corresponding to the stub object was sent as part of a *remoteConstructor* call that was broadcasted to all hosts.

In the new version, just one stub is created when the distributed object is created, on the machine that initiated the object construction (if this machine is the same with the object's owner, then no stub is created at all).

This means that is often possible for a host to receive a remote call containing a global ID that does not have an associated local object (either because the stub has been collected or because it has never been created).

In this case, the Kernel must be able to create the stub on the fly, but the problem is that it does not know what is the Java class that should be instantiated for this purpose. Our solution to this problem was to include additional information in the global ID, so that the JSDA Runtime could decide based on it the type of the object to be created.

We decided that including the whole class name in the global ID would be wasteful (will highly increase the network overhead). Therefore we decided to use just a numerical ID instead of the class name; this ID represents the host ID of the object's owner. A special message containing the object ID needs to be sent to the owner, who will return the class name for the specified object.

This message can be piggybacked since a *dirty* call needs to be sent to the owner upon stub creation anyway. Therefore, this change does not involve any additional network traffic. The overall traffic is at most equal to the amount of data that we would have used if the class name were sent as part of the initial request, since only the class names that are needed are sent (as opposed to sending *all* the class names for the passed

parameters, as part of the initial call – there is no way to know in advance which one will be needed on the destination host).

Figure 3.5.2.1 below shows the steps that need to be done – according to the rules that we just mentioned – once a remote call packet is received:

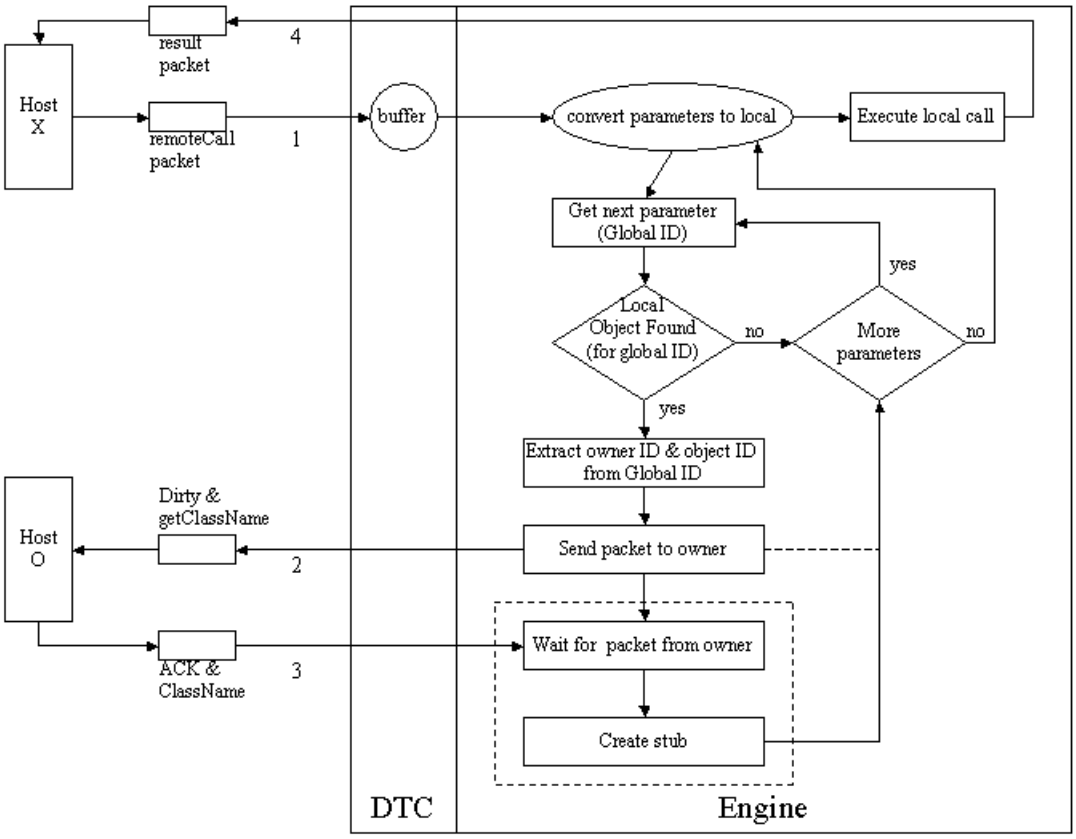


Figure 3.5.2.1 Stub creation

The mechanism illustrated by the Figure 3.5.2.1 above was implemented and tested during *Phase 1B*. However, the last version of JSDA is slightly modified – the blocks in Figure 3.5.2.1 surrounded by a dashed line correspond to actions that will be

executed in a separate thread (the DGC Control thread – see Section 3.5.2.3 for more details). This ensures that if *n* stubs need to be created, this will *not* be done in a serial way – the current application thread will not have to block for each ACK packet it needs to receive. Instead, the packets containing the ACKs and the class names are received by another thread.

3.5.2.2 Weak versus Soft References

We investigated the specific differences between soft and weak references (which are not clearly specified in Sun's *javadoc* documentation) and discovered that softly reachable objects survive several garbage collection cycles while weak references are collected immediately. (This was also confirmed by an article [M98] found on Sun's website, which said that "the only real difference between a soft reference and a weak reference is that the garbage collector uses algorithms to decide whether or not to reclaim a softly reachable object, but always reclaim a weakly reachable object")

We used this – apparently small – difference when we designed the DGC algorithm. As mentioned earlier, we decided to use Soft/Weak references as following:

1. Weak References are used in:
    a. the EntryMaster (WEAK) records in the Object Table
    b. the *WeakLocalToGlobal* hash table
2. Soft References are used for stubs in the Object Table

The references to stubs are soft because we do not want the stubs to be immediately collected. It is possible that future remote calls will want to reuse the stubs; if the stub is still in the object table when an incoming remote call needs it, then we save both CPU time and network traffic, since we do not need to send the *dirty+getClassName* message to the object's owner.

The same thing does not apply to master objects since we never recreate them; therefore we would just waste memory by referring them via soft references.

The fact that stubs are reachable via weak references in the hash table and via soft references in the Object Table has the following implication: if at some moment in time the stub is no longer strongly reachable, it will *first* become unreachable in the hash table and *then* it will become unreachable in the Object Table. Whenever a remote call packet is received, we must ensure that if a stub is found locally then the corresponding entry in the hash table is not empty (if it is, we re-insert the stub in the hash table to ensure consistency).

3.5.2.3 Keeping the Distributed Thread Model Consistent

The distributed thread model in JSDA is based on the following rules:

- A distributed thread in a JSDA environment with *n* hosts consists of *n* individual threads (one running on each host)
- Out of these *n* local threads, at any moment in time, *n-1* threads are in a passive state (waiting for requests) and 1 thread is in an active state (executing).

- All local threads corresponding to a distributed thread share a globally unique thread ID, maintained by the JSDA Kernel.

The two states of a local thread are illustrated if Figure 3.5.2.3A below:

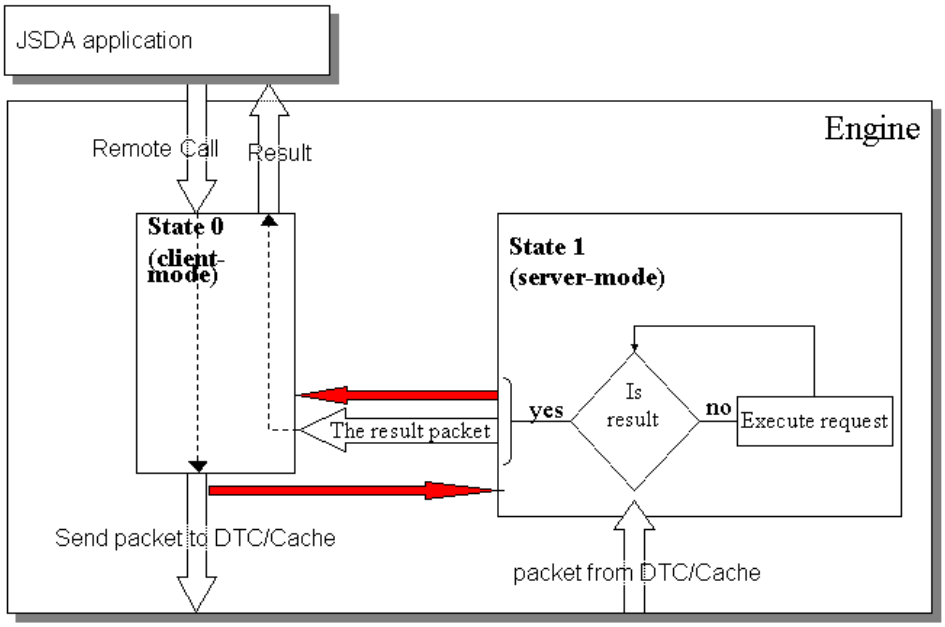


Figure 3.5.2.3A The Two States of A Thread

There are two types of threads in JSDA:

1. Application Threads
2. Kernel Threads

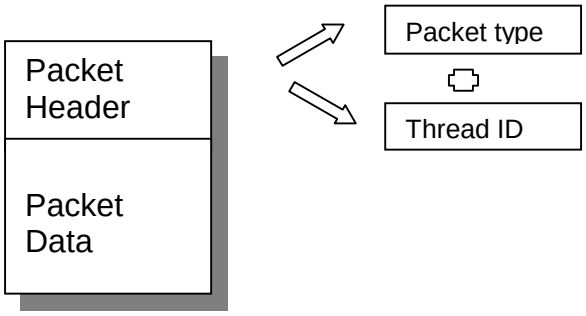
According to the rules specified above, the JSDA Runtime must ensure that a message sent by a remote call initiated by a local thread will be received by an instance of the same distributed thread on the destination host. This was implemented successfully in the first version of JSDA (as part of the DTC module). This section discusses the

changes that we made to the Kernel in order to ensure that the new threads introduced by the DGC module do not corrupt the distributed thread model used for the application threads.

There are two threads created by the DGC:

1. The **Cleane**r – its role is to extract objects from the reference queue (stubs that become softly reachable) and to send clean messages to their owners
2. The **Control** thread – its role it to receive messages sent by Cleaner and all other messages that should not be sent to an application thread. The key idea here is that an application thread that receives a packet that represents anything else than the answer it expects from a remote call, will cause the JSDA Runtime to crash.

Every packet sent by the Kernel has the following structure:



Normally, the DTC module will set the thread ID to the value of the global thread ID corresponding to the current local thread (the current local thread is the one who initiated the request to send the packet). However, it may decide not to do that, in order to preserve the correctness of the algorithm. The choice is made according to the rules below:

Packet *Type*	*Normal*	*Cache*	***Clean***	***Dirty***	***ACK***
Sent by	AppThreadX	CacheWorkX	**Cleaner**	**AppThreadX**	**Control**
Received by	AppThreadX	AppThreadX	**Control**	**Control**	**Control**

Introduced by garbage collection

Table 3.5.2.3B

Here are the reasons behind the new rules:

1. Garbage collection related packages should not be received by the application threads since these threads only expect answers from their remote calls
2. We do not want to assign the Cleaner additional responsibilities, since this will increase the response time and will decrease performance

Figure 3.5.2.3C shows the new structure of the JSDA Kernel, highlighting the thread types used in JSDA:

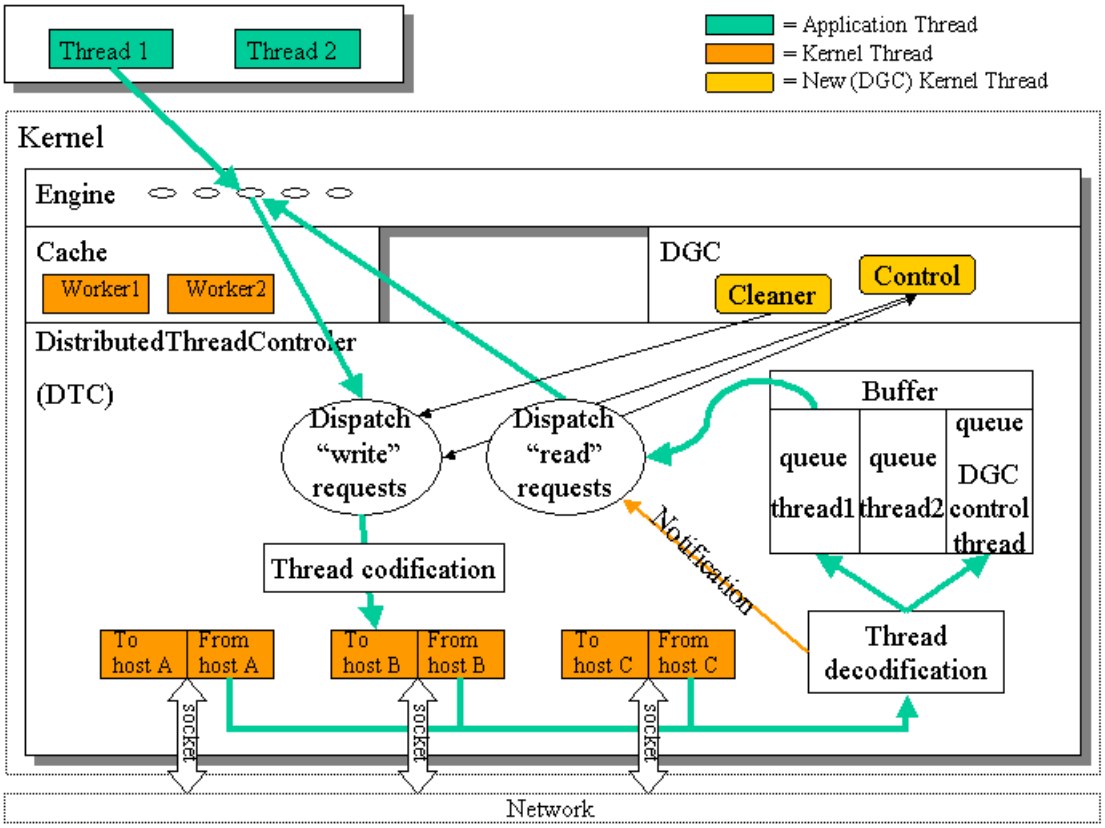


Figure 3.5.2.3C The DGC Threads

When a host receives a remote call that requires the creation of several stubs, one *dirty* message needs to be send to the owner of each stub and one ACK message must then be received, before the result is sent to the caller.

As Table 3.5.2.3B shows, *dirty* messages are sent by application threads (since they are the ones who detect that stubs are needed) but received by the *Control* thread. This implies that the *Control* thread must be able to notify the corresponding application thread once all the ACKs have been received. We implemented this functionality by creating an additional data structure inside the DGC module. Each record in this table stores information about:

1. The application thread ID
2. The number of ACK-s required (the number of stubs newly created)
3. The current number of ACK-s received.

When the values in fields 2 and 3 become equal, the corresponding application thread is notified to send the result to the caller.

3.6 Results

Both the proof and the implementation validated our design for the distributed garbage collection algorithm. Therefore, we can safely say that we proved that the functionality provided by the Reference Objects API is enough to build a distributed collector.

We did not run any performance measurements since the JSDA framework was a prototype itself, and did not allow us to execute full testing against various types of applications. However, our main goal was to investigate the possibility of building a distributed collector (and then to build one) and we succeeded in attaining this goal.

4. Conclusions and Future Work

We designed a distributed garbage collection algorithm for Java. We proved its correctness and also implemented it in a particular framework (JSDA).

As a result of this, we showed that the limited interaction with the local collector offered by the java.lang.ref package is sufficient to build a distributed garbage collector.

As a generalization, we claim that the algorithm can be implemented for any object-oriented language offering features similar to the Reference Objects API.

We estimate that the algorithm is efficient in terms of network and CPU overhead. We explained the decisions that we made in order to keep these overheads to a minimum.

Accurate performance measurements will be desirable as a future step of the evaluation of our approach. The fault tolerance of the algorithm would also be interesting to investigate. Our algorithm does not collect cycles and we believe that this functionality cannot be obtained in Java, which does not allow tracing since pointer references are not accessible from outside the virtual machine.

References

[AFT99] Y.Aridor, M.Factor, A.Taperman – *cJVM: a Single System Image of a JVM on a Cluster.* In Proceedings of the 1999 International Conference on Parallel Processing, 4-11, Sep 1999

[B87] D.I.Bevan – *Distributed garbage collection using reference counting.* In PARLE Parallel Architectures and Languages in Europe, vol.259 of Lecture Notes in Computer Science, 176-186, June 1987

[Bi77] P.B.Bishop – *Computer systems with a very large address space and garbage collection.* MIT Report LCS/TR-178, Laboratory for Computer Science, MIT, May 1977

[BC92] H. Boehm, D.R.Chase – *A Proposal for Garbage-Collector-Safe C Compilation.* Journal of C Language Translation, 126-141, 1992

[BHJL86] A.Black, N.Hutchinson. E.Jul and H.Levy – *Object Structure in the Emerald System.* OOSPLA '86 Proceedings. Sep 1986

[BEN+93] A.Birell, D.Evers, G.Nelson, S.Owicki, E.Wobber – *Distributed Garbage Collection for Network Objects.* TR116, Digital Systems Research Center, Palo Alto, CA, Dec.1993

[BN84] A.D.Birrell and B.J.Nelson – *Implementing remote procedure calls.* ACM Transactions on Computer Systems, 2(1):39-59, Feb 1984.

[Ch84] T.W.Christopher – *Reference count garbage collection.* Software Practice and Experience, 14(6):503-507 Jun 1984

[Cor] http://www.corba.org

[Co91] J.R.Corbin – *The Art of Distributed Applications. Programming Techniques for Remote Procedure Calls.* Springer-Verlag. 1991

[DB76] L.P.Deutch, D.G.Bobrow – *An efficient incremental automatic garbage collector.* Communications of the ACM, 19(7), July 1976

[DCOM] *Component Object Model (COM), DCOM and Related Capabilities.* http://www.sei.cmu.edu/str/descriptions/com_body.html

[DL+78] E.W.Dijkstra, L.Lamport, A.J.Martin, C.S.Scholten, E.F.M.Steffens – *On-the-fly Garbage Collection: An Exercise in Cooperation.* Communications of the ACM, 21(11):965-975, Nov.1978

[DTH99] A.Dancus, R.Teodorescu, R.Handorean – *Java Support for Distributed Applications (JSDA).* Graduation Thesis. Politehnica University of Bucharest (PUB). Romania. July 1999

[Far] Distributed Systems Groups, Technion – *Publications on Fargo.* http://www.dsg.technion.ac.il/ fargo/publications

[F95] M.Fuchs – *Garbage Collection on an Open Network.* International Workshop on Memory Management, Kinross, UK, Sep. 1995

[G89] B.Goldberg – *Generational Reference Counting: A reduced-communication distributed storage reclamation scheme*. *ACM* Proceedings of the SIGPLAN '89 Conference on Programming language design and implementation

[J00] R.Jones – *Directions for Distributed Garbage Collection.* Microsoft Research, Cambridge, UK, Aug 2000

[JL96] R.Jones, R.Lins – *Garbage Collection – Algorithms for Automatic Dynamic Memory Management*. Wiley 1996

[JSp] Sun Microsystems – *JavaSpaces Technology.* http://java.sun.com/products/javaspaces/

[L92] R.D.Lins – *Cyclic reference counting with lazy mark-scan.* Information Processing Letters, 44(4) :215-220, 1992

[Lin] http://www.cs.yale.edu/Linda/linda.html

[LQP92] B.Lang, C.Quenniac, J.Piquer – *Garbage Collecting the World.* ACM Symposium on Principles of Programming, Albuquerque, 1992

[M98] M.Pawlan – *Reference Objects and Garbage Collection*, article available at htpp://developer.java.sun.com, Aug. 1998

[Orc] http://www.cs.vu.nl/vakgroepen/cs/orca.html

[P96] J.M.Piquer – *Indirect Distributed Garbage Collection: Handling Object Migration.* ACM Transactions on Programming Languages and Systems, Vol.13, No.3, Sep 96

[PS95] D.Plainfosse, M.Shapiro – *A Survey of Distributed Garbage Collection Techniques*, International Workshop on Memory Management, Kinross, UK, Sep. 1995

[RMI98] Sun Microsystems – *Java Remote Method Invocation Specification.* Rev.1.50 Oct. 1998

[R98] H.Rodrigues, R.Jones – *A Cyclic Distributed Garbage Collector for Network Objects.* International Workshop on Distributed Algorithms (WDAG), 1996

[Voy] Object Space – *Voyager*. http://www.objectspace.com/voyager

[W92] P.R.Wilson – *Uniprocessor Garbage Collection Techniques.* International Workshop on Memory Management, St.Malo, France, Sep 1992

[W94] P.R.Wilson – *Uniprocessor Garbage Collection Techniques.* Technical Report, University of Texas, Jan. 1994